



Dear Customer:

Thank you for purchasing Intel's 80386 Software Development Package for installation on an IBM PC AT or PC XT (DOS release 3.0 or greater). A checklist is attached so that you can confirm receipt of all items ordered.

Please be aware that by opening this package you have agreed to abide by the Intel license agreement contained in this package. You may make one copy of the licensed program for backup or archival purposes.

For this particular release, we have added two additional tools for your convenience. OH386 converts Intel386™ Object Module Format into hexadecimal format, allowing you more flexibility when burning PROMs. The program development templates simplify processor programming, especially if you are not familiar with the architecture for the Intel386 family of components.

Intel is dedicated to providing you with advanced tools that significantly boost programmer productivity, lower software development costs, and minimize product development times. For more information on Intel products, contact your local sales office.

Intel Corporation
Systems Group

**CHECKLIST FOR MATERIAL INCLUDED WITH
INTEL'S 80386 SOFTWARE DEVELOPMENT PACKAGE**

ITEM	DESCRIPTION
1	Binder containing: <i>Intel386™ Family System Builder User's Guide</i> <i>Intel386™ Family Program Development Templates</i> Intel Software License and Registration Certificate Problem Report Forms Two sets-Product Release Notes One 3-1/2" diskette - DOS 80386/System Builder, Mapper, Binder & Librarian, Additional Utilities One 3-1/2" diskette - DOS 80386/Program Development Templates One 5-1/4" diskette - DOS 80386/System Builder & Mapper One 5-1/4" diskette - DOS 80386/System Binder & Librarian One 5-1/4" diskette - DOS 80386/Additional Utilities One 5-1/4" diskette - DOS 80386/Program Development Templates
2	Binder containing: <i>Intel386™ Family Utilities User's Guide</i>





product release notes

RLL386 VERSION 1.2

OH386

OH386 is a DOS-tool that converts files from the bootloadable Intel386™ Object Module Format into either the 8086 hexadecimal format or the 80386 hexadecimal format. OH386 also has a DEBUG option that includes file header information as part of the output. The OH386 files include the following:

OH386.EXE	the OH386 command file.
SAMPLE	an example file in absolute object format.
HEX32.TXT	a description of hexadecimal file format.
README.TXT	a description of the OH386 program.

SYNTAX

You can run OH386 from the operating system using the syntax:

```
OH386 obj_file [DEBUG] [386] [> hex_file]
```

Where:

OH386	is the command name.
<i>obj_file</i>	is the object file to be converted.
DEBUG	includes the file header information in a readable format.



386

changes the output hex file to 80386 hexadecimal format instead of the default 8086 hexadecimal format.

> *hex_file*

creates the specified file and directs the hexadecimal output to it.

EXAMPLES

In the following example, the file SAMPLE is converted from object module format to 80386 hexadecimal format.

```
OH386 SAMPLE 386 >SAMPL386.HEX
```

In the next example, the same file is converted to 8086 hexadecimal format.

```
OH386 SAMPLE >SAMPL86.HEX
```

In the following example, the object file is converted to 386 format but output is sent to the screen by default.

```
OH386 SAMPLE 386
```

In the next example, the object file is converted to an 80386 hexadecimal file with debug information.

```
OH386 SAMPLE DEBUG 386 >SAMPLE.HEX
```


The following example shows the header information that is included when the DEBUG option is used.

header byte - b2
area of memory covered by records - 000f009b
DATE of CREATION - 11/13/87
TIME of CREATION - 16:39:38
MODULEs CREATOR - 80386 SYSTEM BUILDER, V1.2
GDT BASE - 00011000 GDT LENGTH - 00ff
IDT BASE - 00011100 IDT LENGTH - 0087
TSS SELECTOR - 0068

PARTITION 1

ABSTXT LOCATION - 00000068
LAST LOCATION - 00002421
NEXT PARTITION - 00000000
OSINFO - 00000000

code start address - 000111E0
code length - 00000068

:02000002111ECD
:1000000000000000000000000000000000000000F0
:1000100000000000000000000000000000000000E0
:1000200000000000000000000000000000000000D0
:1000300000000000000000000000000000C0000000A8
:1000400000000000000000000000000000CB000000C300000022
:10005000CB000000CB000000CB000000CB00000074
:08006000000000000000000000000098





product release notes

RLL386 RELEASE 1.3 ON DOS SYSTEMS

These Product Release Notes are divided into the following sections:

- **TOOL-SPECIFIC NOTES:** BLD386, BND386, LIB386, MAP386
- **DOS-SPECIFIC NOTES**
- **MANUAL UPDATE**

BLD386 V1.4

THE CHECKSUM IN BOOTLOADABLE FILES IS INCORRECT

BLD386 produces bootloadable or loadable object files. The default output is a bootloadable object file. The checksum is the last byte in every bootloadable object file. The checksum is the complement of the sum of all the preceding bytes in the object module in 8-bit arithmetic. This checksum is incorrect.

For more information regarding the bootloadable object module format please see the *Intel386™ Family System Builder User's Guide*, Appendix B, "Simple Bootloadable Files in OMF386".



RESERVED DESCRIPTOR TABLE SLOTS CAN BE OVERWRITTEN

BLD386 places default descriptors in the GDT and LDT. Slot 0 in both tables always contains a null descriptor. These cannot be overwritten. In addition, BLD386 automatically places a GDT alias descriptor and an IDT alias descriptor in slots 1 and 2 of the GDT. BLD386 automatically places an LDT alias descriptor in slot 1 of every LDT. These alias descriptors can be overwritten using a build file TABLE definition. However, BLD386 will not warn you if you overwrite the default alias descriptors.

IN FLAT MODEL, ORIGINAL DESCRIPTORS REMAIN IN THE OBJECT FILE

Flat model is the simplest scheme. The program is in one linear address space. There is one code segment called `__phantom_code__`, and one data segment called `__phantom_data__`. By default, `__phantom_code__` and `__phantom_data__` share the same 4-gigabyte overlapping and unprotected address space.

Flat model combines original input segments into the `__phantom_code__` and `__phantom_data__` segments. By default, BLD386 places the `__phantom_code` and `__phantom_data__` descriptors in the GDT with `DPL=0`.

Descriptors for the original segments are invalid because the original segments have been combined into the `__phantom__` segments. The symbols are no longer based on original descriptors. All symbols in original segments are now based on the `__phantom__` descriptors.

BLD386 continues to place original segment descriptors in the LDT. These original descriptors appear in the PRINT file and in the output object module. To eliminate the original descriptors from the output object module, specify NOT CREATED in a build file TABLE definition for the LDT.

For more information regarding flat model, see the *Intel386™ Family System Builder User's Guide*, Chapter 4 (FLAT control) and the *Intel386™ Family Program Development Templates* booklet.



MODULES WITH NO DATA SEGMENT

For example, BLD386 fails if the compiled object code from C programs with no static variables is input directly to the builder. BLD386 cannot combine a STACK segment with a DATA segment of type NORMAL if the NORMAL segment is also EXTERNAL. To avoid this problem, either use BND386 to bind the module before sending it to BLD386 for processing, or ensure that at least one static variable is defined in the program.

BND386 V1.3

USING THE SEGSIZE CONTROL

Reducing the size of a segment to zero by using the SEGSIZE control does not produce Warning 133 as expected. The segment map does not list a segment with size zero.

USING THE RESERVE CONTROL

Using the RESERVE control to reserve large blocks of selector values can produce an output file which MAP386 is unable to process. For example, MAP386 enters an infinite loop when the BND386 controlfile a.cf contains the control rs(0 to 1200). Be careful when using the RESERVE control to reserve large blocks of selectors if you intend to process the output file with MAP386.

LIB386 V1.1

USING PAGELENGTH

The maximum PAGELENGTH is 64K bytes. Specifications greater than 64K bytes are processed <value> MOD 64K bytes.

RUNNING LIB386 INTERACTIVELY

When running LIB386 interactively, a value must always be specified for the PAGELENGTH entry in the SET command. A carriage return does not give the default value, but generates an error.



MAP386 V1.1

Object files containing multiple "linkable" modules cannot be generated simply by concatenating object files. Concatenated object files cause MAP386 to enter an infinite loop.

NOTES FOR ALL RLL386 V1.3 PRODUCTS

<Ctrl>Z is not allowed in a control file.

A blank line in a command within a control file causes all subsequent lines within the command to be ignored. Use the ampersand (&) to continue command lines.

Error 114 indicates a shortage of memory. Ensure that you have the following:

LIB386	512K bytes
MAP386	512K bytes
BND386	512K bytes
BLD386	512K bytes



DOS OPERATION-SPECIFIC NOTES

DIRECTING OUTPUT TO A FULL DISK

When directing output to a full disk, an Intel translator or RLL tool may terminate prematurely without giving an error message. Before invoking these tools, make sure your disk has sufficient space to contain any output they may generate.

FILE-SHARING CONFLICTS

File-sharing conflicts may occur when using an Intel translator or RLL tool in a network environment. Before invoking an Intel translator or RLL tool (with network support), invoke the DOS V3.0 or later SHARE command. Intel recommends that you invoke the SHARE command in your AUTOEXEC.BAT file.

USING <CTRL><BREAK> INSTEAD OF <CTRL>C

<Ctrl>C does not work as the interrupt character like <Ctrl><Break> does. Use <Ctrl><Break> instead of <Ctrl>C.

FILES NAMED WITH AN 8-DIGIT HEXADECIMAL NUMBER

Files named with an 8-digit hexadecimal number with no extension may be left in the current :WORK: directory after typing <Ctrl><Break> to abort an Intel translator, RLL tool, or a user program converted to DOS with UDI2DOS.EXE. Programs create these temporary files to store intermediate data. The temporary files are deleted at the end of the program's normal execution. You can delete or ignore the files. They contain no important information.



FATAL ERROR WHEN SPECIFYING LONG PATHNAMES

A fatal error occurs when specifying excessively long pathnames for output files. When such an error occurs, the translator or RLL tool aborts. While the DOS manual indicates that the maximum number of characters in a pathname is 63, in practice various products restrict pathnames to fewer than 63 characters. To ensure compatibility with all products, make sure that all output pathnames do not exceed 43 characters.

ERROR #21 --- FILE DOES NOT EXIST

The operating system may issue this message even though the file does exist. The DOS operating system is installed incorrectly. Re-install the operating system and make sure that it is DOS V3.0 or later. DOS V3.0 or greater has a different COMMAND.COM file.

INPUT/OUTPUT FILE CONTENTS MAY BE LOST

The contents of a file may be lost when the same file is specified for input and output to a translator or RLL tool. Depending on the order in which a translator or RLL tool opens the input and output files, the file may be overwritten if it is used for both input and output. When a file is opened for output on DOS, previous contents are lost unless the file is already opened for input and the SHARE command is in effect. When invoking a translator or RLL tool under DOS, ensure that the output filename differs from the input filename.



MANUAL UPDATE

The following corrections and additions apply to the *Intel386™ Family System Builder User's Guide*.

Page 1-9, last sentence

Gate descriptors in the interrupt descriptor table can also be task gates. Please change the wording as follows:

Trap, task, and interrupt gate descriptors are stored in the interrupt descriptor table (IDT).

Pages 2-14 and 5-2, in Figure 2-2 and Figure 5-1
BLD386 can also produce an errorprint file as output.

Pages 3-27 and A-7

In the syntax for *makepages* everything following *dirseg-id* is optional, denoted by square brackets []:

$$\text{makepages} \Rightarrow \text{dirseg-id} \left[\left(\left\{ \begin{array}{l} \text{maketables} \\ \text{bits} \\ \text{LOCATION} = \text{public-id} \end{array} \right\} [, \dots] \right) \right]$$

Page 3-29

Please add the following note after the Discussion title:

NOTE

The order of options after declaring a *dirseg-id* is important. The *maketables* option, if present, must precede any *bits* option. Only the *bits* option can be repeated. The LOCATION option can occur in any position in the list.

Page 3-31, in Example 2

Please add a semicolon after the RESERVE specification as follows:

MEMORY	--two page tables and one
(RESERVE = (800000h .. 0FFFFFFFh));	-- directory built for
	-- 8M bytes of memory,



Pages 3-45, in Table 3-5

A gate-id can also be a task gate in the IDT.

Page 3-46

In the description for *buf-id*, please change the wording as follows:

is a public symbol attached to a buffer where BLD386 can store the base address and limit of the newly defined table. BLD386 places the base address and the limit of the table being defined...

Page 3-47, in Figure 3-7

A selector value is the offset in bytes from the base of the descriptor table; a slot number is the position of the descriptor in the table; the GDT alias is in slot number 1; the IDT alias is in slot number 2:

3 to n	18h to nh	(user-specified descriptors)
2	10h	Alias Segment Descriptor for IDT
1	8	Alias Segment Descriptor for GDT
0	0	All zeros (null descriptor)

Slot	Selector Value	GDT
------	----------------	-----

Page 3-48, in Figure 3-8

A selector value is the offset in bytes from the base of the descriptor table; a slot number is the position of the descriptor in the table:

2 to n	10h to nh	(user-specified descriptors)
1	8	Alias Segment Descriptor for LDT
0	0	All zeros (null descriptor)

Slot	Selector Value	LDT
------	----------------	-----

Page 3-51, in Example 2

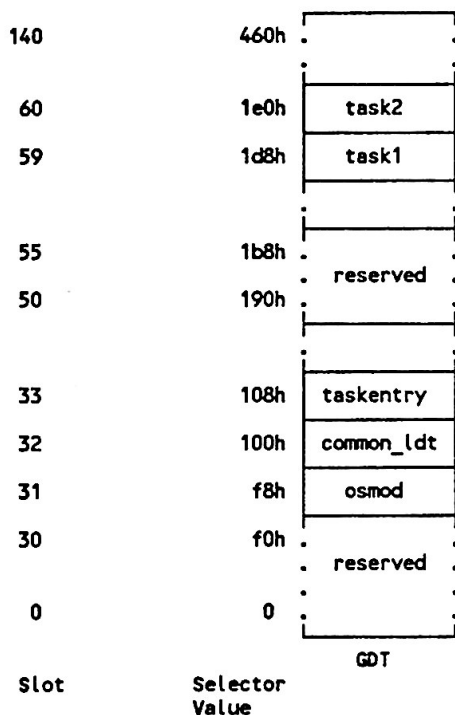
Please change the comment for the ENTRY specification:

```
ENTRY = (osmod,                --the GDT has 141 total slots because
        common_ldt,            -- 80 extra are added after the last
        taskentry,              -- descriptor is entered
        59:(task1, task2)),
```



Page 3-51, in Figure 3-9

A selector value is the offset in bytes from the base of the descriptor table; a slot number is the position of the descriptor in the table; the GDT has 141 total slots:



Page 5-13, in the last paragraph

Please change the wording as follows to add information about errorprint files:

...Other messages appear at the end of the print file. Messages can be repeated in an errorprint file (see [NO]ERRORPRINT control in Chapter 4). Warnings and non-fatal error messages can be suppressed in the print file and the errorprint file (see [NO]WARNINGS control in Chapter 4).

Pages 6-1 through 6-20

These pages are superseded by the *Intel386™ Family Program Development Templates* booklet.



Page F-2

Please add underscoring to denote operator input:

Username: SYSTEM <cr>

Password: _____ <cr>

\$ SET DEFAULT SYSSUPDATE <cr>

\$ @VMSINSTAL RL386 <cr>

Page F-3

Please add underscoring to denote operator input:

XVMSINSTAL-W-DECNET, Your DECnet network is up and running.

* Do you want to continue anyway [NO]? y

* Are you satisfied with the backup of your system disk [YES]? y.

* Where will the distribution volumes be mounted [MSA0:] <tape-device>

Please mount the first volume of the set on <tape-device>

* Are you ready? y

.
. .
. .

* Do you want to purge files replaced by this installation [YES]? y

This kit contains an Installation Verification Procedure to verify the correct installation of the VAX/VMS RL386 tools.

* Do you want to run the IVP after the installation [YES]? y

* Enter where the RL386 V1.3 images directory INTEL386 should be created: <device-name>

Page F-4

Please add underscoring to denote operator input:

\$ CREATE /DIRECTORY SYSSSPECIFIC:[SYSHLP.EASE]

Page F-6

Instead of the list given, please refer to the *Intel386™ Family Program Development Templates* booklet for the names of files in the ease of use kit.





Development Solutions

Intel386™ Family
System Builder
User's Guide



Intel386™ Family System Builder User's Guide

Notational Conventions

The notational conventions described below are used throughout this manual.

UPPERCASE	Characters shown in uppercase must be entered in the order shown but may be entered in either uppercase or lowercase.
<i>italics</i>	Italics indicate a metasympol that may be replaced with an item that fulfills the rules for that symbol. Metasymbols in tables are not always shown in italics.
<i>filename</i>	This represents a character string recognized by the operating system to identify a file, including the device-name and directory or pathname where necessary.
[]	Brackets enclose optional arguments or parameters.
{ }	Braces indicate that one and only one of the enclosed entries must be selected unless the entire field is also surrounded by brackets, in which case choosing an entry is entirely optional.
{ } ...	Braces followed by an ellipsis indicate that at least one of the enclosed entries must be selected unless the entire field is also surrounded by brackets, in which case choosing an entry is entirely optional. The items may be used in any order unless otherwise noted.
	The vertical bar separates options within brackets [] or braces { }.
...	An ellipsis indicates that the preceding item may be followed by other like items; the items must be separated by at least one space unless the items begin with a separating character such as a slash, a parenthesis, or a comma.
[, ...]	Brackets enclosing a comma and an ellipsis indicate that the preceding item may be followed by other like items and the items must be separated by commas.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown.
< >	The angle brackets indicate the key described within; for example, <return>.
⇒	The "derives" symbol indicates that the meta-symbol on the left is more specifically defined as the item on the right.

INTEL386™ FAMILY SYSTEM BUILDER USER'S GUIDE

Order Number: 481342-001

REV.	REVISION HISTORY	DATE
-001	Original Issue.	8/88

In the United States, additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

Above	iLBX	Intellink	MICROMAINFRAME	Ripplemode
BITBUS	im	iOSP	MULTIBUS	RMX/80
COMmputer	IMDDX	iPAT	MULTICHANNEL	RUPI
CREDIT	iMMX	iPDS	MULTIMODULE	Seamless
Data Pipeline	Inboard	iPSC	ONCE	SLD
ETOX	Insite	iRMK	OpenNET	SugarCube
FASTPATH	Intel	iRMX	OTP	UPI
Genius	intel	iSBC	PC BUBBLE	VL5iCEL
A	Intel376	iSBX	Plug-A-Bubble	376
i	Intel386	iSDM	PROMPT	386
i ² iCE	intelIBOS	iSXM	Promware	386SX
ICE	Intel Certified	KEPROM	QueX	4-SITE
iCEL	Intelevision	Library Manager	QUEST	
iCS	Intelligent Identifier	MAPNET	Quick-Erase	
iDBP	Intelligent Programming	MCS	Quick-Pulse Programming	
iDIS	Inteltec	Megachassis		

VAX, MicroVAX, VMS, and MicroVMS are trademarks of Digital Equipment Corporation.

Copyright © 1988, Intel Corporation. All Rights Reserved



Preface.....	ix
--------------	----

Chapter 1 Overview

1.1	The Intel386™ Family Utilities and BLD386.....	1-1
1.2	Execution Environment of Intel386™ Family Processors.....	1-5
1.2.1	Addressing Using Segments	1-5
1.2.2	Gates	1-9
1.2.3	Tasks	1-10
1.3	System Development Using BLD386	1-12

Chapter 2 Introducing BLD386

2.1	Functional Characteristics.....	2-1
2.2	Address Assignment.....	2-3
2.2.1	Logical Address Assignment.....	2-3
2.2.2	Privilege Checking.....	2-4
2.2.3	Automatic Gate Creation	2-5
2.3	Memory Definition and Address Assignment.....	2-5
2.4	Descriptor Table Creation	2-6
2.4.1	Global Descriptor Table	2-7
2.4.2	Interrupt Descriptor Table	2-7
2.4.3	Local Descriptor Table	2-7
2.5	Descriptor Manipulation and Creation	2-8
2.5.1	Segment Descriptors	2-8
2.5.2	Special System Data Segment Descriptors.....	2-9
2.5.3	Control-Transfer Descriptors.....	2-9
2.6	Task State Segment Creation	2-10
2.7	Page Table Creation	2-10
2.8	Reference Resolution and Type Checking	2-11
2.9	Exportation	2-11
2.10	Aliases	2-12
2.11	Descriptor Relocation Information	2-12
2.12	Debug Information.....	2-12
2.13	Input and Output	2-13

Chapter 3 Build Language Specifications

3.1	The Build Language.....	3-1
3.2	The Build Program.....	3-5
	Program Definitions	3-7
3.3	Build Program Examples	3-62

Chapter 4 Invocation

4.1	DOS Invocation Syntax	4-1
4.2	VMS Invocation Syntax.....	4-2
4.3	Console Messages.....	4-4
4.4	Control Files	4-5
4.5	BLD386 Controls	4-6
	Command Entries.....	4-13

Chapter 5 Input and Output

5.1	File Types	5-1
5.2	Print File.....	5-3
	5.2.1 Header.....	5-3
	5.2.2 Build Program Listing	5-4
	5.2.3 Segment Map.....	5-4
	5.2.4 Gate Table	5-7
	5.2.5 Task Table	5-8
	5.2.6 Page Tables.....	5-12
5.3	Warning and Error Messages	5-13

Chapter 6 System Building Examples

6.1	Flat Model Templates for DOS	6-1
	6.1.1 The Flat Memory Model	6-2
	6.1.2 The Flat Model Templates	6-3
	6.1.3 The Print File.....	6-18
6.2	Using 80386 Call Gates with C.....	6-21
	6.2.1 The Call and the Return	6-21
	6.2.2 Stack Cleanup.....	6-22

Appendix A Syntactical Summary

A.1	The Build Language Syntax.....	A-1
A.2	The Build Program Syntax.....	A-2

Appendix B Simple Bootloadable Files in OMF386

B.1	Overview of Bootloadable File Structure.....	B-1
	B.1.1 Simple OMF386 Bootloadable File Structure	B-1
	B.1.2 Bootloadable Module Header.....	B-2
	B.1.3 Bootloadable Partition.....	B-3

B.2	Bootloadable Structure with Selected Examples.....	B-5
B.2.1	Bootloadable Module Header Example.....	B-5
B.2.2	Bootloadable Partition with ABSTXT Example.....	B-7
B.3	Annotated Object File Hex Dump.....	B-8

Appendix C Error Messages

C.1	System-Level Exceptions.....	C-2
C.2	Invocation or Input Object Exceptions	C-2
C.3	Build File Messages.....	C-42
C.4	Internal Processing Exceptions.....	C-69

Appendix D Master List of Reserved Words

Appendix E Master List of Controls and Abbreviations

Appendix F VMS Software Installation

F.1	Installation Media.....	F-1
F.2	Execution Environment	F-1
F.3	Installation Requirements	F-1
F.4	Installation Procedure	F-2
F.5	RL386 Installation Files.....	F-5
F.6	RL386 Commands and the Help Facility.....	F-5
F.7	Ease of Use Kit.....	F-6

Glossary

Index

Service Information	Inside Back Cover
---------------------------	-------------------

Figures

1-1	Primary Functions of Intel386™ Family Relocation, Library, and Linkage Utilities.....	1-3
1-2	Additional Functions of Intel386™ Family Relocation, Library, and Linkage Utilities.....	1-4
1-3	Intel386™ Family Processor Registers Used in Addressing	1-6
1-4	Segment Descriptor Format.....	1-7
1-5	Selector Format	1-7
1-6	Example Address Translation.....	1-9

1-7	Gate Descriptor Format	1-10
1-8	Task State Segment Format	1-11
2-1	Protected Mode Privilege Rules	2-4
2-2	BLD386 Input and Output.....	2-14
3-1	Aliasing Scheme with Items of Different Sizes.....	3-8
3-2	Gate Descriptor Format	3-17
3-3	Example Memory Layout	3-26
3-4	Page Directory and Page Table Entries	3-29
3-5	Segment Descriptor Format	3-35
3-6	Data/Stack Combined Segment Organization	3-38
3-7	Default Table Organization for GDT	3-47
3-8	Default Table Organization for LDT	3-48
3-9	GDT Entries for Example 2.....	3-51
3-10	80386 TSS Format	3-55
3-11	Data/Stack Combined Segment Organization	3-57
3-12	EFLAGS Register.....	3-58
4-1	80386 Flat Memory Model	4-29
5-1	BLD386 Input and Output.....	5-2
5-2	BLD386 Print File Header.....	5-3
5-3	BLD386 Build Program Listing	5-4
5-4	BLD386 Print File Segment Map	5-5
5-5	BLD386 Print File Gate Table	5-7
5-6	BLD386 Print File Task Table	5-9
5-7	BLD386 Page Tables	5-12
6-1	Builder's 80386 Flat Memory Model.....	6-2
6-2	Simple Flat Model System Memory Map.....	6-20
B-1	Formatted Hex Dump of Bootloadable Module Header	B-5
B-2	Formatted Hex Dump of Partition with ABSTXT Record1	B-7

Tables

2-1	Descriptor Table Entries.....	2-6
3-1	Build Language Elements	3-2
3-2	BLD386 Build Language Reserved Words, Keywords, and Abbreviations.....	3-4
3-3	Syntax Summary for Gate Definition	3-18
3-4	Syntax Summary for Segment Definition	3-40
3-5	Valid Entries for Tables	3-45
3-6	Syntax Summary for Alias Descriptor for Table Segment	3-47
3-7	Entry-List Parameters for the GDT	3-49
3-8	Entry-List Parameters for the IDT.....	3-49
3-9	Entry-List Parameters for an LDT.....	3-50
3-10	Syntax Summary for Task State Segment	3-57

3-11	Syntax Summary for Task Descriptor.....	3-60
4-1	Summary of BLD386 Controls for DOS.....	4-8
4-2	Summary of BLD386 Controls for VMS.....	4-10
4-3	Standard Abbreviations for BLD386 Controls	4-12
B-1	Bootloadable Object File Structure.....	B-2
B-2	Bootloadable Module Header.....	B-2, B-6
B-3	GDT and IDT Fields.....	B-3
B-4	Bootloadable Partition.....	B-3
B-5	Bootloadable Partition Table of Contents.....	B-4, B-7
B-6	Bootloadable Absolute Text (ABSTXT) Section	B-4, B-8



This manual describes how to use the 80386 System Builder (BLD386). BLD386 is a tool for configuring system software for the Intel386™ family of microprocessors. BLD386 is used to:

- Create and modify descriptor tables, segment and system descriptors (including gates), and task state segments.
- Create page tables and directories for use in paged memory systems.
- Assign physical addresses to segments and descriptor tables.
- Configure system interface files for use in developing application programs.

Use BLD386 with the other 80386 Utilities: the 80386 Binder (BND386), the 80386 Librarian (LIB386), and the 80386 Mapper (MAP386). Use BND386 to create combined, linkable modules from separately translated modules. Use the LIB386 interactive utility to organize linkable object modules into libraries, and, after libraries are created, to access, add, delete, or replace individual modules. Use MAP386 to generate various maps from information in object modules.

This manual addresses system designers and programmers who are familiar with system and operating-system design principles, and 80386 programming languages and translators. Knowledge of the Intel 80x86 architecture is important. You should also be familiar with the rules governing the interfaces between modules written in different languages.

Manual Organization

This manual is organized as follows:

- Chapter 1 describes many of the architectural and Intel-specific terms used in this manual.
- Chapter 2 describes the major functions performed by BLD386.
- Chapter 3 describes the build language. The syntax and implications of the definitions are discussed in detail.

- Chapter 4 describes BLD386's invocation, console messages, and controls.
- Chapter 5 describes the file types associated with input and output, and describes the output print file.
- Chapter 6 presents system building examples with invocation, source code, and explanations.
- Appendix A formally states the syntax of a build program.
- Appendix B describes the object module format of a bootloadable file created by BLD386.
- Appendix C lists all BLD386's warning and error messages with the causes, effects, and possible recovery actions.
- Appendix D lists all reserved words, parameter keywords, and abbreviations for the build language.
- Appendix E lists all controls and their abbreviations for BLD386.
- Appendix F is a guide to installing the software for VMS.
- The Glossary defines some of the terminology used in this manual.
- The Index identifies pages which define and use key terms and concepts.

Related Publications

Refer to the following publication for related information:


- *Intel386™ Family Utilities User's Guide*, order number 481343
- *ASM386 Assembly Language Reference Manual*, order number 480251
- *Introduction to the 80386*, order number 231252
- *80386 Programmer's Reference Manual*, order number 230985
- *80386 System Software Writer's Guide*, order number 231499

Contents

Chapter 1 Overview

1.1	The Intel386™ Family Utilities and BLD386.....	1-1
1.2	Execution Environment of Intel386™ Family Processors.....	1-5
1.2.1	Addressing Using Segments	1-5
1.2.2	Gates	1-9
1.2.3	Tasks	1-10
1.3	System Development Using BLD386	1-12






The 80386 System Builder, BLD386, is one of four Intel386™ family utilities. Use BLD386 to create the data structures for Intel386 family software, and to assign addresses to code and data for a bootloadable system or a loadable object module. This chapter describes how to use BLD386 in developing software along with the Intel386 family utilities. It also introduces the execution environment of the Intel386 family processors and the protected-mode data structures.

1.1 The Intel386™ Family Utilities and BLD386


Intel386 family software varies from simple single-task systems to complex multitask systems that execute at several privilege levels. BLD386 can be used to configure system software statically (before run-time).



Whether your software is an application program which requires minimal operating system support, or a program which depends extensively on the services of the operating system, or all or part of an operating system itself, the Intel386 family utilities provide total program development and system configuration resources for Intel386 family software. The utilities are:

- LIB386, the 80386 Librarian
- MAP386, the 80386 Mapper
- BND386, the 80386 Binder
- BLD386, the 80386 System Builder

BND386 and BLD386 are tools for linking and configuring Intel386 family software. LIB386 and MAP386 provide additional resources to aid in program development and debugging. Figure 1-1 shows the most common uses of the utilities. Figure 1-2 shows other uses of the utilities.



80386 and 80286 translators translate source files into object modules. When building a system, collect these object modules according to their common functions and characteristics, and combine them with

BND386. Then use BLD386 to process the modules and add the initial protected-mode data structures. Section 1.2 describes the data structures.

Input to BLD386 is usually one or more translated and linked object modules and a build program file with definitions for the protected-mode data structures. Output from BLD386 is usually a bootloadable module.

A bootloadable module is a single- or multiple-task object module with absolute addresses assigned to all entities. This module is used for system cold-start. The object module includes task information and tables created according to the build file definitions. A bootloadable module can be loaded by a simple bootstrap loader, or it can be the foundation for a ROM-based system.

BLD386 also accepts an input control file containing controls and names of input modules or files. It can produce a listing of the build program; maps of segments, tables, and tasks; and an error listing. BLD386 supports modular system development by exporting selected data structures and even entire modules to a linkable file. Optionally, BLD386 can create a dynamically loadable output module.

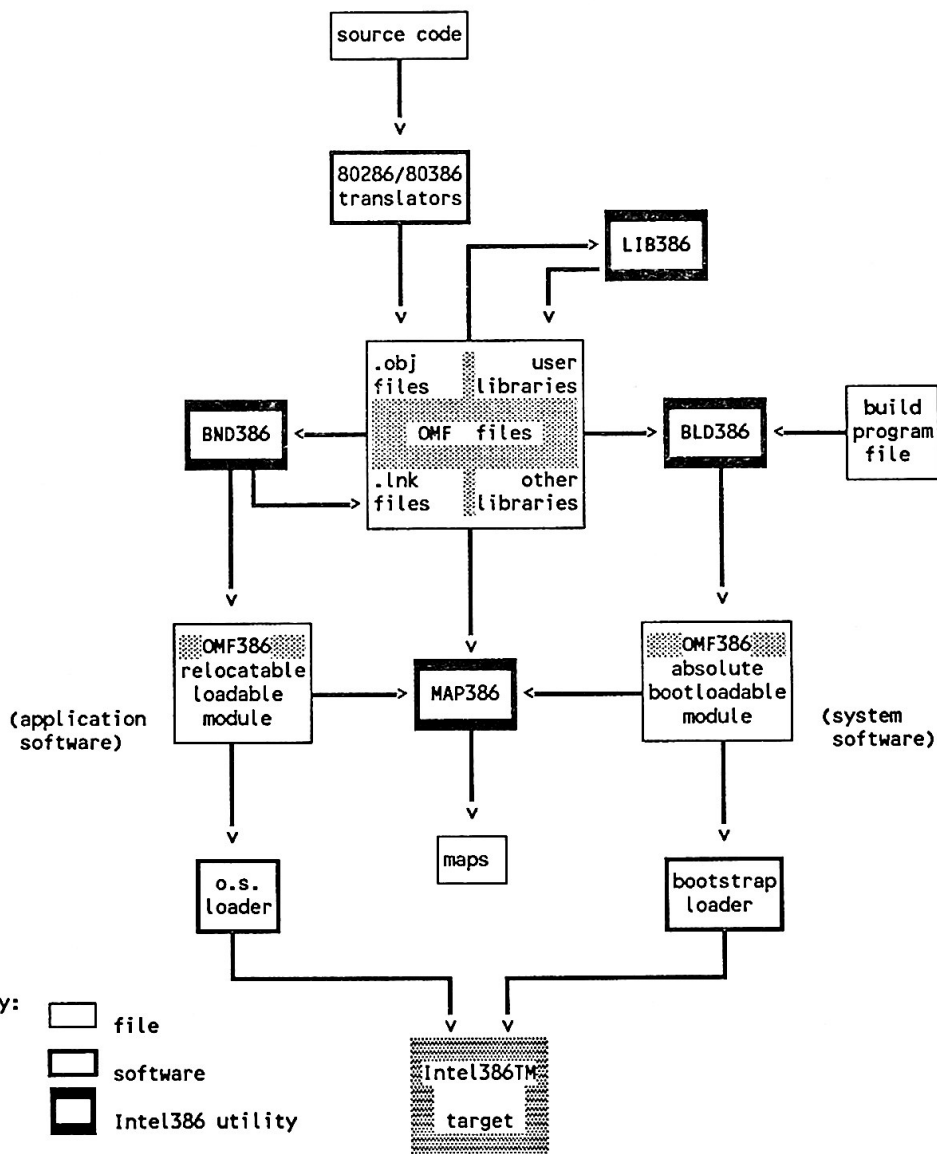


Figure 1-1 Primary Functions of Intel386™ Family Relocation, Library, and Linkage Utilities

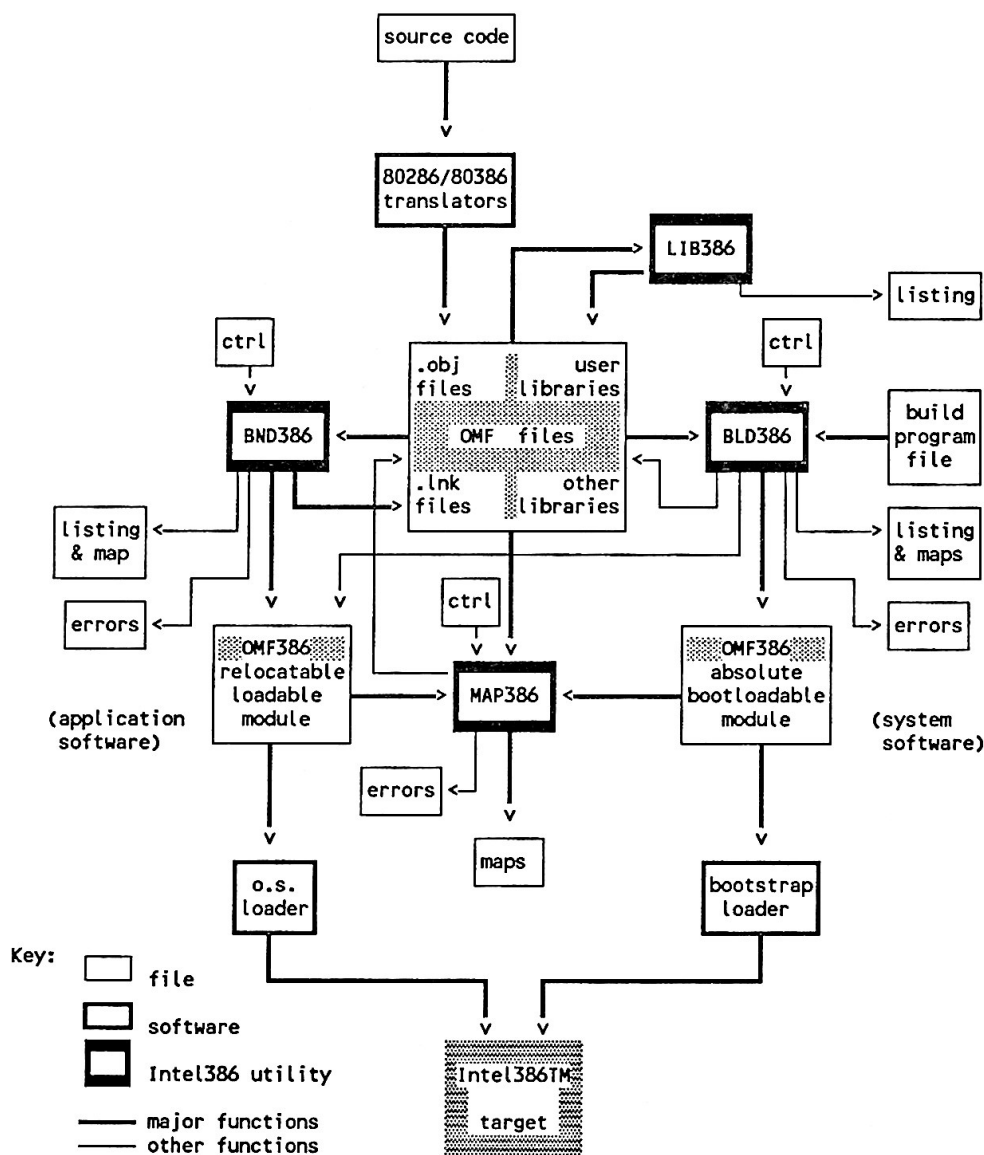


Figure 1-2 Additional Functions of Intel386™ Family Relocation, Library, and Linkage Utilities

1.2 Execution Environment of Intel386™ Family Processors

This section describes features of the Intel386 family architecture and the related data structures for program and system development.

1.2.1 Addressing Using Segments

Figure 1-3 shows the Intel386 family register sets that are used in address calculation. The 32-bit registers include 8 general registers including ESP, the stack pointer; a status register; and an instruction pointer. When the instruction pointer or a general register is used for addressing a target instruction or operand, it contains the offset of the target relative to a segment base address.

On an Intel386 family processor memory is partitioned logically. The processor expects one base address for code, and one for data. There is often a separate base address for the stack. Associating a size limit with each base address defines a segment.

The data structure that stores the defining base and limit of a segment is called a descriptor. Additional attributes that define the type of the segment and the way it can be accessed are also stored in the descriptor. The format of a general segment descriptor is shown in Figure 1-4.

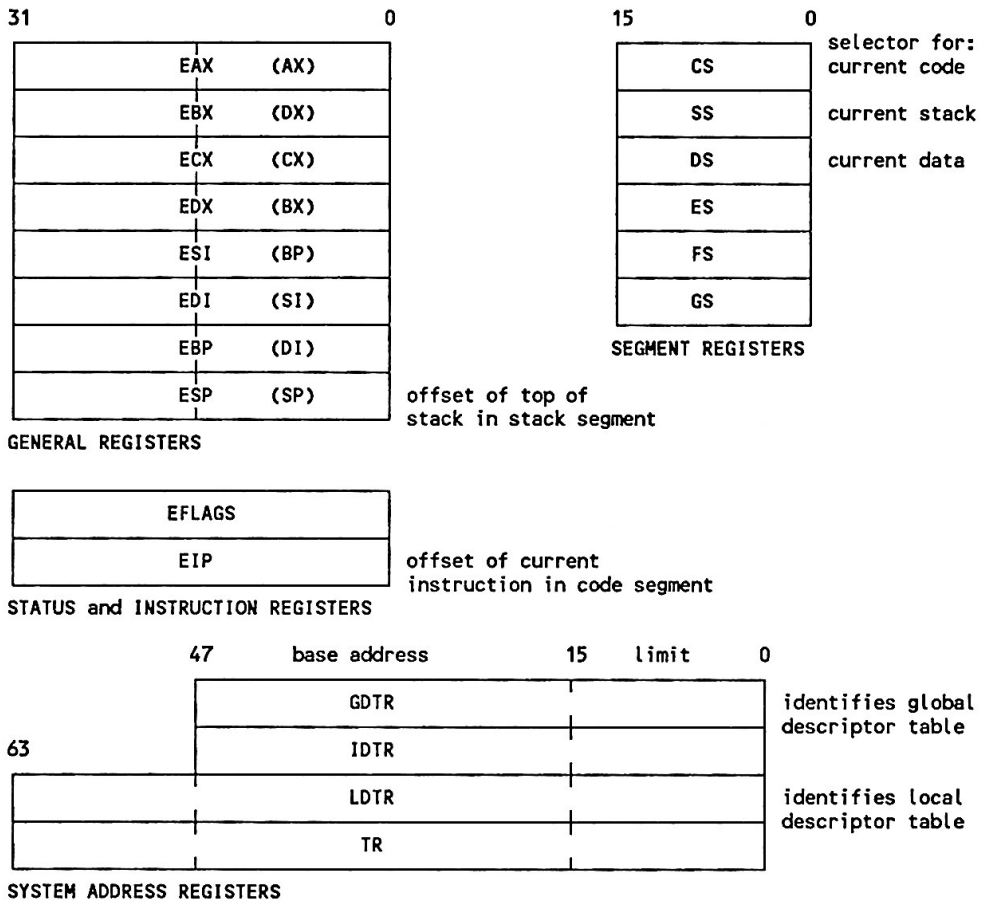


Figure 1-3 Intel386™ Family Processor Registers Used in Addressing

The Intel386 family processors access segment descriptors in descriptor tables. Each table is a segment containing is an array of descriptors. The base addresses of the global descriptor table (GDT) and the current local descriptor table (LDT) are stored in the system address registers, GDTR and LDTR. The Intel386 family processors do not require a local descriptor table to run (see Section 1.2.3, Tasks).

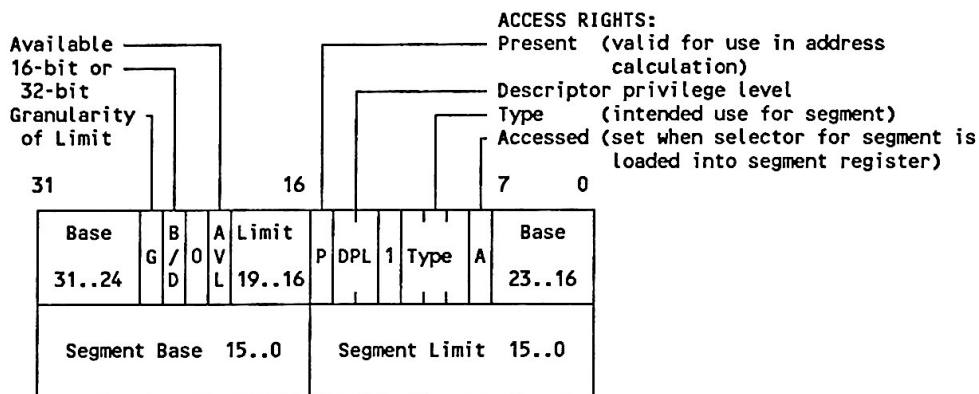


Figure 1-4 Segment Descriptor Format

To compute an address, the processor uses a selector as an index for the appropriate descriptor in the global or local descriptor table. Selectors for currently active segments are stored in the segment registers, CS, DS, ES, FS, GS, and SS. It is the programmer's responsibility to ensure that the segment selector is loaded into a segment register before the execution of an instruction with an operand not in the current segment. The ASM MOV instruction or a task switch loads the segment registers. The ASM END directive can specify initial values for the CS, SS, and DS segment registers. The format for a selector is shown in Figure 1-5.

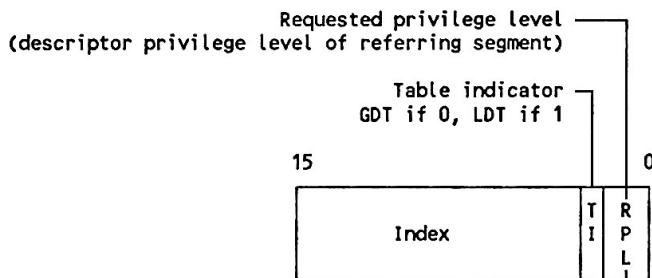


Figure 1-5 Selector Format

A base address and an offset comprise a logical address. The corresponding linear address is the sum of the base and the offset. The Intel386 family processor's memory management unit automatically performs this address translation. This is a simplified description of the process:

- The instruction operand yields the offset.
- The instruction type determines which selector to use.
- The selector determines which descriptor table to use.
- The system address register yields the base address of the descriptor table.
- The selector indexes the target segment's descriptor within the descriptor table.
- The segment descriptor yields the base address of the target segment.
- The offset is added to the base address of the target segment.

The actual implementation makes use of hidden cache registers which speed the translation process. Figure 1-6 represents a simplified example of address translation. The resulting linear address is the same as the physical address unless paging is enabled. Paging is not available on the 80376.

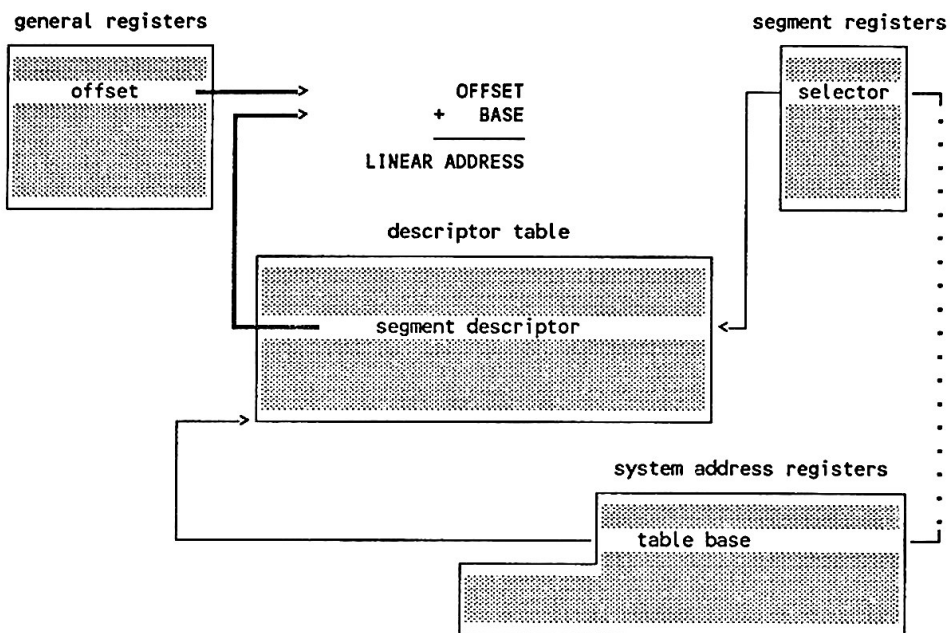


Figure 1-6 Example Address Translation

1.2.2 Gates

An Intel386 family processor uses special descriptors called gates to regulate transfer of control between executable segments at different privilege levels. Each gate is a special form of a descriptor. There are four kinds of gate descriptors:

- call gates
- task gates
- trap gates
- interrupt gates

Call and task gate descriptors are stored either in the global descriptor table (GDT) or in a local descriptor table (LDT). Trap and interrupt

gate descriptors are stored in the interrupt descriptor table (IDT). The format for a gate descriptor is shown in Figure 1-7.

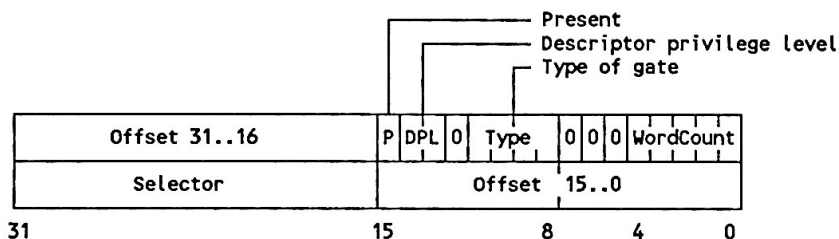


Figure 1-7 Gate Descriptor Format

Access to a segment protected by a gate requires that the control transfer be made to the gate descriptor before entry is made. The privilege level of the gate and the accessibility of the table in which it is stored offer protection from unrestricted access.

1.2.3 Tasks

A task is a collection of one or more code segments at the same privilege level together with the data and stack segments necessary to complete a well-defined function during execution. The Intel386 family processors do not require a task to run. Three different data structures describe a task: a task gate descriptor, a task state segment (TSS), and a TSS descriptor (or task descriptor).

The task gate descriptor (described in Section 1.2.2) provides indirect, protected reference to a task state segment. The task state segment contains the context and state of the processor at the time of a task switch. The format of a task state segment is shown in Figure 1-8. The TSS descriptor describes the task state segment itself. A TSS descriptor is a general segment descriptor.

A task gate descriptor can be stored in the GDT, in an LDT, or in the IDT if the system designer chooses to make exception handlers separate tasks. The selector portion of a task gate descriptor points to

1-11

Figure 1-8 Task State Segment Format

1.3 System Development Using BLD386

This section outlines a scenario for system development using the Intel386 family utilities. Although it is not a complete strategy for developing software, this model does show how BND386 and BLD386 can be used in system development. A basic, multilevel, protected-mode system can be designed and developed from separately translated and linked subsystems. Each subsystem is a group of tasks which operate at the same privilege level. Early in the design process, the following design characteristics are established:

- What memory model the system will function in: flat (non-segmented), segmented, paged, or a hybrid
- Which code and data are allocated at each privilege level
- How interlevel references are to be resolved
- Which higher-privileged code is to be made visible to other code at less-privileged levels
- How this visibility is to be provided (e.g., using gates or conforming segments)
- What part of the code and data is to be shared at the same privilege level
- What entry points into the system should exported to be made accessible to other programs

After the design phase is completed, the separate modules of each subsystem are written and translated. The different subsystems are then configured from the separately translated modules. BND386 links the modules that eventually form a subsystem at a specific privilege level. Hence, for a four privilege-level system, BND386 must be used at least four times, once for each privilege level.

When each subsystem has been linked, the designer assigns the access and protection attributes to each subsystem. A build program is created to specify the characteristics of and relationships among the subsystems. A build program contains definitions in the following categories:

Segment attributes	establish the privilege levels, access rights, etc., of subsystems and other entities. Because BLD386 does not combine segments, referring to individual input segments by name is an unambiguous way to manipulate segment attributes.
Gates	are entry points to subsystems for access by less-privileged subsystems.
Descriptor tables	are defined based on the expected code- and data-sharing among the subsystems (in the global descriptor table), as well as code and data that should be protected from other tasks (in local descriptor tables). An interrupt descriptor table contains gates that point to interrupt procedures or tasks. Build program definitions specify the tables themselves, reserve slots in them, and install descriptors.
Exported entities	might be used to define a system interface. BLD386 optionally places the gates, public symbols, and/or segments that provide the interface into a separate (linkable) export file. Application programmers then use export files to create the interface between applications programs and system software.
Aliases	establish one or more additional names for certain named entities, such as segments, tables, or tasks. Aliases are descriptors that reside in descriptor tables. They allow the same segment to be accessed in different ways.

- Memory definition** defines which memory ranges are to be used for segments, tables, or tasks; reserves ranges of memory to prevent them from being allocated; and defines the read/write rights for ranges of memory.
- Task state segments** are created which define the initial code, data, and stack segments for a task, initial status of registers, flags, and the LDT associated with the task.
- Page tables and directory** are optionally created. The segment or segments in which they reside are identified, and the entries for the tables and the directory are defined. Paging is not available on the 80376.

BLD386 reads the build program file and follows the definitions to configure a loadable or bootloadable output module.

Contents

Chapter 2 Introducing BLD386

2.1	Functional Characteristics.....	2-1
2.2	Address Assignment.....	2-3
2.2.1	Logical Address Assignment.....	2-3
2.2.2	Privilege Checking.....	2-4
2.2.3	Automatic Gate Creation	2-5
2.3	Memory Definition and Address Assignment.....	2-5
2.4	Descriptor Table Creation	2-6
2.4.1	Global Descriptor Table.....	2-7
2.4.2	Interrupt Descriptor Table	2-7
2.4.3	Local Descriptor Table.....	2-7
2.5	Descriptor Manipulation and Creation	2-8
2.5.1	Segment Descriptors	2-8
2.5.2	Special System Data Segment Descriptors.....	2-9
2.5.3	Control-Transfer Descriptors.....	2-9
2.6	Task State Segment Creation	2-10
2.7	Page Table Creation.....	2-10
2.8	Reference Resolution and Type Checking.....	2-11
2.9	Exportation	2-11
2.10	Aliases	2-12
2.11	Descriptor Relocation Information	2-12
2.12	Debug Information.....	2-12
2.13	Input and Output	2-13



2.1 Functional Characteristics

Use the definitions in the build file (see Chapter 3) and controls in the invocation line or control file (see Chapter 4) to determine what a BLD386 invocation does. If a build file is not used, BLD386 uses default definitions. BLD386 performs the following major functions:

- Assigns physical addresses to entities; sets segment limits depending on the USE16, USEREAL, or USE32 attribute; and sets access rights with the SEGMENT definition.
- Creates descriptor tables with the TABLE definition: one global descriptor table (GDT), one interrupt descriptor table (IDT), and zero or more local descriptor tables (LDTs).
- Allows multiple descriptors for the same entity with different attributes with the ALIAS definition.
- Creates and manipulates descriptors in the descriptor tables with the ALIAS, CREATESEG, SEGMENT, TASK, and TABLE definitions.
- Creates a bootloadable module or a dynamically loadable module with the [NO]BOOTLOAD control.
- Overrides segmentation and configures memory as one continuous linear address space with the FLAT control.
- Selects required modules from specified libraries automatically, as needed to resolve symbolic references.
- Produces a file containing the build program listing, a summary of build processing, and the initial state of the system (see the LIST, MAP, and PRINT controls in Chapter 4, and Chapter 5).
- Detects and reports errors and warnings found during processing (see the ERRORPRINT control and Section 4.2 in Chapter 4, and Chapter 5).
- Suppresses all or specified warnings and errors except for fatal errors with the NOWARNINGS control.

- Creates 80286 or 80386 gates for inter-level transfers, interrupt processing, and tasks with the GATE definition.
- Creates task state segments (TSSs) and task gates for multitask applications with the TASK and GATE definitions.
- Fills or suppresses filling uninitialized areas of segments with a byte value with the [NO]FILL control.
- Performs type checking across symbols of the same name with the TYPE control.
- Creates a page directory and one or more page tables for memory defined as being present, and assigns physical addresses to linear addresses with the PAGING definition and the PAGETABLES control.
- Fills Block Symbol Storage (BSS) areas of segments with zeros.
- Allows for memory addresses to wrap from high to low addresses within a segment. During memory allocation, address assignments are adjusted accordingly.
- Creates new segments (primarily for stacks) with the CREATESEG definition.
- Allows memory ranges to be reserved or allocated for specific entities with the MEMORY definition.
- Creates object files containing exported system entries and conforming segments with the EXPORT definition.
- Processes or removes debug information from output modules with the [NO]DEBUG control.
- Creates information for relocating GDT references in loadable output modules with the NOBOOTLOAD and RELDESC controls.
- Issues messages to guide creation of 80376 output module with the MOD376 control.

NOTE

Because BLD386 processes the build file definitions in a single pass and does no forward-referencing, all information needed to comply with a build file definition must be available to BLD386 when it processes that definition. An entity referenced in a build file definition must either exist in input modules, or must already have had its build file definition processed.

2.2 Address Assignment

BLD386 automatically performs the following for all input modules:

- Assigns logical and absolute addresses.
- Checks privilege levels.
- Creates call gates for calls between segments at different privilege levels.
- Fills BSS (Block Symbol Storage) with zeros.

NOTE

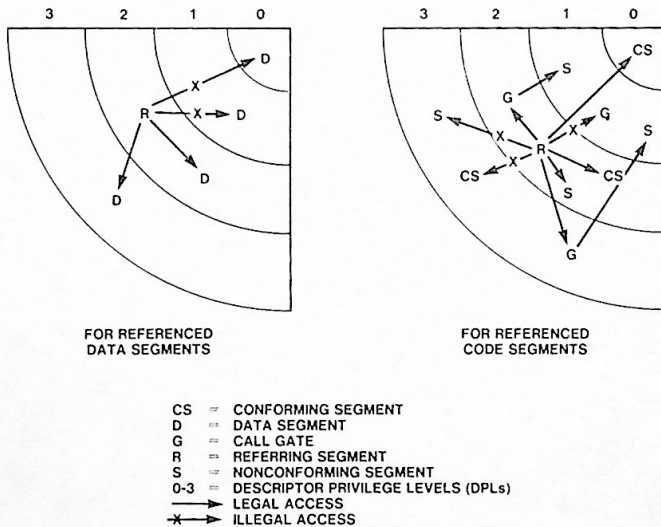
Unlike BND386, BLD386 is not a linking tool. BLD386 does not combine segments, except to create a data/stack combined (DSC) segment if it encounters data and stack segments with the same combine name in one module. (See the SEGMENT definition in Chapter 3 of this manual, as well as the *Intel386™ Family Utilities User's Guide* for additional information on segment combination.)

2.2.1 Logical Address Assignment

BLD386 creates descriptors for segments and system structures. These descriptors contain the logical addresses that BLD386 assigns. With the USEREAL (8086 compatible) attribute in effect, the descriptor contains the paragraph number of the segment base; this value is the base of the segment divided by 16. Code segments must be USE32 for the 80386.

2.2.2 Privilege Checking

BLD386 checks intersegment references to determine whether the access is in accord with the privilege rules shown in Figure 2-1. Only data segments at least as privileged as the executing code are available to the code. Access to another code segment of the same privilege as the executing code does not require a gate. Access to a less-privileged code segment is not permitted. Access to a more-privileged code segment can be made through a gate at least as privileged as the executing code. A gate is not required to access a more-privileged code segment which is "conforming" (see the SEGMENT definition in Chapter 3).



121934-4

For code to refer to a data segment, the data segment must not be more privileged than the code. Code segment R can refer to another code segment directly if they have the same privilege level. For code segment R to refer to a more-privileged code segment, either the access must be through a gate at least as privileged as code segment R, or the accessed code segment must be "conforming".

Figure 2-1 Protected Mode Privilege Rules

2.2.3 Automatic Gate Creation

BLD386 automatically creates a gate to regulate access to another segment at a different privilege level if the `USEREAL` attribute has not been specified (see the `SEGMENT` definition, Chapter 3), if the `FLAT` control has not been specified (see Chapter 4), and if the target satisfies the following conditions:

- The target is in a code segment.
- The target code segment is at a more privileged level than that of the referring segment.
- The target is not referenced in a user-defined gate of an appropriate privilege level.
- The target is 80286- or 80386-compatible; i.e., an 80386 call gate must address an 80386 code segment. All gates must be 80386-type for an object module to run on an 80376 target.

2.3 Memory Definition and Address Assignment

For a bootloadable file, BLD386 assigns absolute addresses to all tables, task state segments (TSSs), and segments. For tables and TSSs, the alignment is "paragraph" and this cannot be overridden; however, absolute addresses can be specified in a build file `TABLE` or `TASK` definition. For segments, the default alignment is set by the language translator; this alignment can be overridden in a build file `SEGMENT` definition. Absolute addresses can also be specified for segments in a build file `SEGMENT` definition. See Chapter 3 for details on the build file definitions.

BLD386 first assigns addresses to items that have a base address specified in a build file definition. BLD386 then assigns default addresses to tables, tasks, and segments that do not have a base address specified. Default addresses start at the lowest free address in available memory. Memory addresses can wrap from high to low addresses within a segment.

Use the `RESERVE`, `RANGE`, and `ALLOCATE` specifications within the `MEMORY` definition in the build file to optionally define memory (see Chapter 3).

BLD386 checks for possible conflicts; e.g., two items that are not aliases assigned to the same address, or a read/write segment placed in a ROM region.

2.4 Descriptor Table Creation

BLD386 creates a GDT and an IDT, and it can create multiple LDTs. BLD386 installs descriptors into these tables. Descriptors can be installed in a user-specified order, reserving table slots, in a TABLE definition in the build file (see Chapter 3).

The types of descriptors that can be installed in each descriptor table are summarized in Table 2-1.

Table 2-1 Descriptor Table Entries

Entry	GDT	LDT	IDT
Segment descriptors	yes	yes	
TSS descriptors	yes		
LDT descriptors	yes		
Call gates	yes	yes	
Task gates	yes	yes	
Interrupt gates			yes
Trap gates			yes

BLD386 provides several automatic descriptor table features, described in Sections 2.4.1, 2.4.2, and 2.4.3. See the TABLE definition in Chapter 3 for more information.

2.4.1 Global Descriptor Table

Although user-designated descriptors can be installed in the GDT, BLD386 automatically installs in the GDT any TSS or LDT descriptors defined but not explicitly installed.

BLD386 always creates a GDT with its first descriptor all zeros. BLD386 creates and installs two table-alias descriptors: one that points to the GDT and one that points to the IDT. These descriptors define the GDT and the IDT as writable data segments. An operating system uses table-alias descriptors for manipulating the GDT and the IDT.

2.4.2 Interrupt Descriptor Table

BLD386 always creates a default IDT with 32 slots when creating a bootloadable module. Although the size can be changed and the contents of this default IDT can be overwritten in a build file TABLE definition (see Chapter 3), the first 32 slots correspond to Intel-defined or Intel-reserved interrupt vectors.

2.4.3 Local Descriptor Table

In each LDT, BLD386 installs a table-alias descriptor that defines the LDT as a writable data segment. This allows an operating system to manipulate the LDT dynamically. Use the build file TABLE definition (see Chapter 3) to overwrite this descriptor.

If any segment descriptors, call gates, or task gates are defined but not explicitly installed, BLD386 installs them in the first defined LDT or creates an LDT to house them if none exists.

2.5 Descriptor Manipulation and Creation

BLD386 creates segment descriptors and system descriptors. Segment descriptors are required for code, data, and stack segments. System descriptors include:

- Special system data segment descriptors:
 - LDT descriptors
 - TSS descriptors
- Control-transfer descriptors:
 - Call gates
 - Task gates
 - Interrupt gates
 - Trap gates

2.5.1 Segment Descriptors

BLD386 creates segment descriptors. By default, BLD386 extracts descriptor values from input module information. Use the build file `SEGMENT` definition (see Chapter 3) to change the following descriptor fields:

- Present bit
- Descriptor privilege level (DPL)
- Access rights
- Base address
- Limit
- Alignment
- 8086-, 80286- or 80386-type (For stacks, this indicates whether the push/pop instruction is 16- or 32-bit; for a code segment, this indicates whether instructions are 16- or 32-bit versions. Some instructions behave differently if they are 16- or 32-bit versions.)

Use the `CREATESEG` definition (see Chapter 3) to create non-executable segments such as stack segments at build time. Also see the `SEGMENT` definition in Chapter 3 for information on creating data/stack combined segments.

2.5.2 Special System Data Segment Descriptors

BLD386 creates TSS and LDT descriptors while creating TSSs and LDTs. Use the TASK and TABLE definitions (see Chapter 3) to override defaults for the following fields of these descriptors:

- Present bit
- DPL
- Base address
- Limit

2.5.3 Control-Transfer Descriptors

Use the GATE definition (see Chapter 3) to create any of the following four types of gates:

- Call gate
- Interrupt gate
- Task gate
- Trap gate

When a gate is specified in the build file but its type is not defined, a default call gate is generated. The gate's type, 80286 or 80386, is made the same as that of the addressed segment. For all gates, use the GATE definition (see Chapter 3) to control the following fields:

- Name of the gate
- Type of gate
- Entry point (destination selector and offset)
- Descriptor privilege level (DPL)
- Present bit
- Word count
- 80286- or 80386-type

Only 80386-type gates are valid for use on an 80376 target.

2.6 Task State Segment Creation

BLD386 automatically creates a TSS for the main module when no task definition is provided in the build file. The initialization values are extracted from the main input module.

Use the TASK definition (Chapter 3) to create TSSs (task state segments) and to control the following TSS attributes:

- Name of the task
- TSS descriptor
- Task LDT selector
- Initial code and data segments (code segment and data segment selectors, and instruction pointer)
- One to three stack selectors corresponding to privilege levels 0, 1, and 2
- I/O privilege level setting, interrupt status, debug flag, and virtual mode setting
- Base address
- Size of TSS (anything over the minimum for use of system)

If these fields are not specified individually in a build file TASK definition, BLD386 extracts initialization information from a specified input module.

2.7 Page Table Creation

BLD386 creates one directory and any number of page tables (see the PAGETABLES control in Chapter 4). Use the PAGING definition (see Chapter 3) to control the following attributes:

- Segment location for page directory and page tables
- Bit setting that controls flags in the directory or table entries
- Location for storing the 32-bit absolute address of the page directory
- Where and how much memory is to be paged

The 80376 does not support paging.

2.8 Reference Resolution and Type Checking

BLD386 attempts to resolve all references to external symbols in its input modules. BLD386 searches input modules and specified libraries for public symbols that satisfy external references. References are considered resolved if the public and external symbol names and types match. (Type checking is done only if the TYPE control is specified. For more information, see Chapter 4.) Unresolved references are listed in the print file.

2.9 Exportation

BLD386 can selectively export entry points and other information needed by other tasks or by application programs (see the EXPORT definition in Chapter 3). Exportation makes the following entities available for subsequent use by BND386 or BLD386:

- Gates
- Public procedures
- Segments
- Entire modules

BLD386 places export information into linkable object files that are useful in the following task-definition activities:

- Providing visibility to operating system code for application programs created using BND386
- Providing visibility to GDT-based public symbols
- Providing access to conforming segments
- Making modules available outside the immediate program environment

2.10 Aliases

BLD386 supports aliases to allow one program entity to access another entity which would be illegal or difficult to access otherwise. An alias is a descriptor which refers to the same entity as another descriptor, but probably has different attributes assigned to the entity. To create an alias, specify the name of the entity to be accessed, the alias name, and the new attributes. See the ALIAS definition in Chapter 3.

2.11 Descriptor Relocation Information

For loadable modules, BLD386 generates information that can be used to relocate GDT and LDT descriptors when contention arises during loading. This information is generated only for loadable modules when the RELDESC control is used (see Chapter 4).

2.12 Debug Information

When the DEBUG control is used (see Chapter 4), BLD386 creates debug information for all new entities created by definitions in the build file. This information is used by MAP386 and by symbolic debuggers. Debug information includes the following:

- Gate name and record
- Task name and record
- Segment name and public symbols

BLD386 also optionally removes debug information from output files when the NODEBUG control is used.

2.13 Input and Output

Input and output for BLD386 are described in detail in Chapter 5. BLD386 accepts the following types of input:

- One or more 80286 or 80386 object files. These object files can be produced by either 80286/80386 translators or BND286/BND386 (see Chapter 4).
- One or more 80286 or 80386 object modules from libraries
- A build program using the build language (described in Chapter 3) specifying the objects to be built or exported
- One or more export modules from a previous BLD286 or BLD386 invocation
- Optional control files that contain file names and/or controls which would normally appear on the invocation line

BLD386 provides the following types of output:

- A bootloadable module containing absolute memory images of 80376 or 80386 software
- A dynamically loadable system image
- Export modules containing exported system entry points for other tasks or code at other privilege levels
- A build program listing in the print file
- A summary of the results of segment, gate, and public-symbol processing in the print file
- Listings of errors detected in the invocation, the build program, and/or input modules in the print file
- Fatal error messages appearing at the console

Figure 2-2 shows the types of input and output allowed when using BLD386.

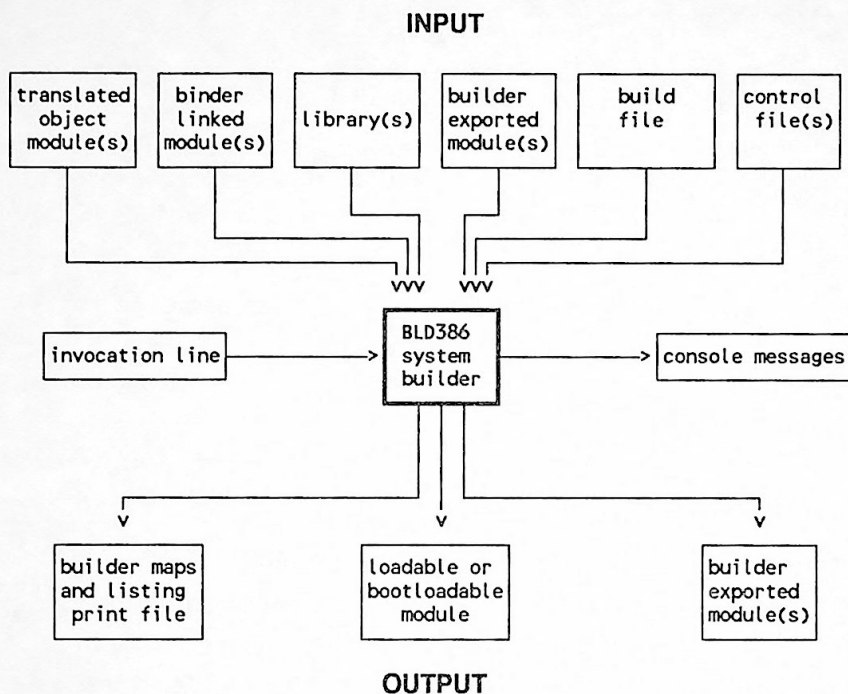


Figure 2-2 BLD386 Input and Output

Contents

Chapter 3 Build Language Specifications

3.1	The Build Language.....	3-1
3.2	The Build Program.....	3-5
	ALIAS Definition.....	3-7
	CREATESEG Definition.....	3-10
	EXPORT Definition	3-12
	GATE Definition	3-15
	MEMORY Definition	3-21
	PAGING Definition.....	3-27
	SEGMENT Definition	3-33
	TABLE Definition	3-43
	TASK Definition.....	3-52
3.3	Build Program Examples	3-62



This chapter describes the elements and syntax of the language for build programs. Each major program element is called a definition. Definitions are presented in alphabetical order. Discussion and examples follow each definition.

3.1 The Build Language

Use the build language to create a system program. The build file contains the build program.

BLD386 uses definitions in the build program along with linkable input modules to create data structures such as descriptors, task state segments (TSSs), page tables, and descriptor tables. BLD386 also uses the definitions to determine memory locations of code and data and to selectively export information to linkable output modules.

The build language is free-format and is not case-sensitive. Use the following characters to construct build language tokens and delimiters:

- The letters A through Z, uppercase and lowercase
- The digits 0 through 9
- The special characters: @ ? _ () : = ; , . - + # *
- The white-space characters: <space>, <tab>, <return>

Tokens are continuous strings of characters. For this discussion, the line-terminator <return> is treated as a single character. Table 3-1 summarizes the syntax for creating build language tokens and delimiters.

Table 3-1 Build Language Elements

Token or Delimiter	Starting Character	Legal Characters	Ending Character
Comment	-- (two hyphens)	printing and white-space	<return>
Identifier	letter @ ? _	letter @ ? decimal digit	last character before delimiter, 40 or fewer characters
Decimal number	decimal digit	decimal digit	optional D or d
Hexadecimal number	decimal digit	decimal digit A through F a through f	H or h
Single character delimiters	() : = ; . - + ,		
Multiple character delimiter	..		
Other delimiters	white-space	white-space	white-space

Build programs can contain comments used for explanatory text. A comment starts with two adjacent hyphens, --, and ends with a line-terminator. Comment text can contain other printing and white-space characters.

Identifiers are names assigned to the build program, to the data structures it creates, to memory ranges, to modules, and to public symbols. These names can be no more than 40 characters long. Identifiers must begin with a letter, @, ?, or _. The rest of an

identifier can contain more of the same characters, as well as decimal digits. In syntax notation, an identifier ends with the suffix *-id*.

Numbers can be expressed in either hexadecimal or decimal form. Hexadecimal numbers consist of the decimal digits 0 through 9 and the letters A through F (upper or lower case), and must have an H or h suffix. The first digit in a hexadecimal number must be a decimal digit (0 through 9). Decimal numbers consist of the digits 0 through 9 with the D or d suffix, or with no suffix at all. In syntax notation, a *number32* is a value from 0 through $2^{32}-1$.

Delimiters separate consecutive identifiers and numbers from one another. The single-character delimiters () : = ; . - + and , and the token .. all have specific places in the syntax of the build definitions. The other delimiters <space>, <tab>, <return>, and the comment only serve to separate tokens.

Table 3-2 lists reserved words, keywords, and abbreviations for use in build language syntax. Some have no abbreviations.

Table 3-2 BLD386 Build Language Reserved Words, Keywords, and Abbreviations

Word	Abbr.	Word	Abbr.
ALIAS	AI	P	
ALIGN	AN	PAGE	
ALLOCATE	AL	PAGED	PD
AT		PAGETABLES	PT
BASE	BA	PAGING	PA
BITSETTING	BS	PARAGRAPH	PH
BYTE	BY	[NOT] PRESENT	[NO] PS
CALL	CA	QWORD	QW
CODE	CO	RAM	
[NOT] CONFORMING	[NO] CF	RANGE	RN
[NOT] CREATED	[NO] CD	[NOT] READABLE	[NO] RA
CREATESEG	CS	RESERVE	RS
DATA	DT	RESET	
[NOT] DEBUGTRAP	[NO] DB	ROM	
DPL		RW	
DRESET		SEGMENT	SM
DSET		*SEGMENTS	
DWORD	DW	SET	
END	DN	SPECLN	SP
ENTRY	ET	STACKS	ST
[NOT] EXPANDDOWN	[NO] ED	SYMBOL	SB
EXPORT	EO	TABLE	TB
GATE	GA	*TABLES	
GDT		TASK	TA
IDT		*TASKS	
[NOT] INITIAL	[NO] II	TRAP	TR
INPAGE	IN	UD1	
[NOT] INTENABLED	[NO] IE	UD2	
INTERRUPT	IT	UD3	
IOPRIVILEGE	IP	US	
LDT		USE16	
LIMIT	LI	USE32	
LOCATION	LA	USREAL	USERL
MEMORY	MO	[NOT] VIRTUALMODE	[NO] VM
NOT	NO	WC	
OBJECT	OJ	WORD	WO
		[NOT] WRITABLE	[NO] WA

3.2 The Build Program

A build program has one or more definitions, as follows:

```
program-id ; {  $\left\{ \begin{array}{l} \text{createseg-definition} \\ \text{segment-definition} \\ \text{gate-definition} \\ \text{task-definition} \\ \text{table-definition} \\ \text{alias-definition} \\ \text{memory-definition} \\ \text{paging-definition} \\ \text{export-definition} \end{array} \right\}$  ; }... END
```

Where:

program-id is an identifier for the build program.
x-definition is defined in this chapter in alphabetical order.

NOTE

Because BLD386 processes the build program definitions in a single pass and does no forward-referencing, all information needed to comply with a definition must be available to BLD386 when it processes that definition. Any entity referenced in any definition must either exist in input modules or must already have had its build-program definition processed. The order indicated above: *createseg-definition*, *segment-definition*, *gate-definition*, etc., is recommended for most build programs, although you may need to use a particular type of definition more than once in a build program.

Build language definitions are described in alphabetical order in the following pages. Information for each definition is provided in the following sequence:

1. The purpose and scope of the definition
2. The syntax of the definition
3. An explanation of each unresolved element in the syntax definition

4. Discussion based on key elements of the definition
5. Summary tables that provide the definition's syntax, valid abbreviations, default values, and relationship to system data structures
6. Figures, where relevant, to provide additional detail
7. Notes that warn of constructions which can cause errors
8. Examples

Section 3.3 of this chapter provides examples of complete build programs.

Purpose and Scope

This definition establishes aliases for specific named entities: segments, tables, and tasks. Establishing such an alias allows access to the same absolute memory location while treating the aliased item as if it were a different entity. Each alias can define different type, access rights, or limit for the segment. The base address must be the same for the alias and the original. Several aliases can be established concurrently for the same item, if necessary.

Syntax

alias-definition \Rightarrow ALIAS *newalias* [, ...]

newalias \Rightarrow *main-id* (*alias-id* [, ...])

Where:

main-id is the name of a segment, table, or task. It must be defined earlier in the build program (see SEGMENT definition, TABLE definition, and TASK definition in this chapter), or it must exist in the input object module, the GDT, or the IDT.

alias-id is a descriptor of a readable and writable (RW) segment name to the aliased *main-id*. The segment must exist in the input object module or in the build program. After *alias-id* is made a new alias of *main-id*, it can be placed in a different slot in a different descriptor table (see TABLE definition in this chapter), although all such slots point to the same absolute memory location.

Discussion

If the output is loadable, BLD386 supplies information concerning both *main-id* and its aliases. The loader can use this information to allocate space and initialize descriptors.

ALIAS Definition (continued)

If the output is bootloadable, BLD386 assigns absolute addresses to the aliased entities.

An *alias-id* must indicate a segment that is readable and writable (RW).

The attributes (e.g., privilege level, base address, limit, and other attributes) for *alias-id* can also be defined in the same build program. (See SEGMENT definition in this chapter.) Individual *alias-ids* that alias one *main-id* can themselves have different attributes. These multiple aliases are also aliased to each other.

NOTE

In Figure 3-1, if the size of the *main-id* segment is less than the size of the *alias-id* segment, the memory space occupied by the remaining portion of the *alias-id* segment is still available to BLD386 for its default memory allocation scheme. Avoid such aliasing schemes.

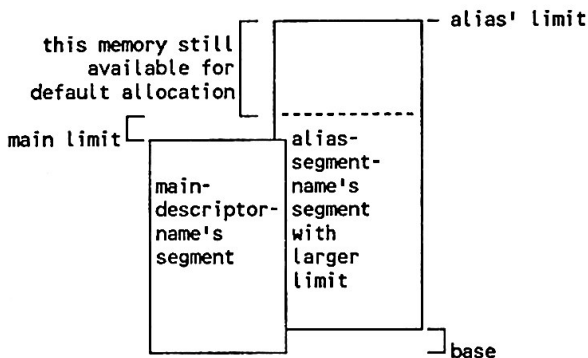


Figure 3-1 Aliasing Scheme with Items of Different Sizes

Example

In this example, a partial build program shows three different main descriptors being assigned aliases.

ALIAS

```
gdt_name (gdt_alias_seg,      --gdt_alias_seg and two
          gdt_alias_dummy,    -- other names are
          gdt_temp),          -- aliased to gdt_name

ldt_name (ldt_alias,          --ldt_alias, ldt_temp
          ldt_temp),          -- are both aliased to
                               -- ldt_name

task_1   (task_2,             --task_2 and two other
          task_3,             -- names are aliased
          task_4);            -- to task_1
```

CREATESEG Definition

Purpose and Scope

This definition creates new non-executable-segment descriptors during the system build. Use CREATESEG to create stack segments for multitask environments without having to code and translate them, or to create stacks beyond those created by a compiler when software is written exclusively in high-level languages.

Syntax

createseg-definition \Rightarrow CREATESEG *newseg* [, ...]

newseg \Rightarrow *segment-id* [(SYMBOL = *symbol-id*)]

Where:

segment-id is the name of the new segment.

symbol-id is the name of the public symbol used by other modules to reference the new segment.

Discussion

The *createseg-definition* makes a USE32 stack segment that has read/write access and privilege level 3. Set or adjust these and other attributes of the segment in a subsequent SEGMENT definition in the build program.

If *symbol-id* is specified, BLD386 creates a public symbol with that name. This symbol's address is offset 0 in the new segment, its type is null, the combine-type is changed to DSC (data/stack combined), the limit is set to 1, and the length of the data portion of this DSC segment is set to 1. See SEGMENT definition in this chapter for more on DSC segments.

NOTES

If *symbol-id* is not specified, the segment has a limit of 0.

An executable segment cannot be created in a build program.

CREATESEG Definition (continued)

If a segment with the same name as *segment-id* already exists, BLD386 issues a warning and does not create a new segment.

If an already-existing public symbol duplicates *symbol-id*, BLD386 issues a warning and the existing public symbol remains unchanged.

Example

In this example, a partial build program shows two uses of the *createseg-definition* to make stack segments.

```
CREATESEG
new_seg (SYMBOL = new_seg_symbol)      --a stack segment named
                                         -- new_seg is created with
                                         -- public symbol
                                         -- new_seg_symbol

temp_seg;                               --a stack segment named
                                         -- temp_seg is
                                         -- created with no public
                                         -- symbol and a limit of 0
```

EXPORT Definition

Purpose and Scope

This definition creates a file containing structures useful for system interfaces. It places structures (modules, gates, symbols, descriptors, procedures, entry points, or segments) in linkable modules for use in subsequent BND386 or BLD386 invocations. These structures must be already defined in the build program, or must be available in an input module.

Syntax

export-definition \Rightarrow EXPORT *send-out* [, ...]

send-out \Rightarrow #*filename* (*export-items* [, ...])

export-items \Rightarrow *export-id* (*export-list*)

export-list \Rightarrow $\left\{ \begin{array}{l} \text{module-id } [, \text{id-list}] \\ \text{id-list } [, \text{module-id}] [, \text{id-list}] \end{array} \right\}$

id-list \Rightarrow $\left\{ \begin{array}{l} \text{gate-id} \\ \text{public-id} \\ \text{segment-id} \end{array} \right\} [, ...]$

Where:

#filename designates the name of the file in which the exported entities are to be placed. The # symbol is required. If the file already exists, a new version is written.

export-id assigns a name to the linkable output module that is to contain the exported information.

module-id designates an entire input module. The exported module contains all entities in the input and any SEGMENT definition specifications in the build program. Debug information, if any, is not exported. Only one *module-id* is permitted per *export-id*.

EXPORT Definition (continued)

gate-id

designates an individual gate. Only the following types of gates should be exported:

- Gates installed in the GDT
- Gates installed in an LDT that point to segments with descriptors in the GDT

public-id

designates a public symbol or entry point in a segment that has a descriptor installed in the GDT.

segment-id

designates a descriptor that is installed in the GDT. The segment descriptor is exported, but not the segment's contents. The contents can be exported by using *module-id*.

EXPORT Definition (continued)

Example

In this example an operating system is being built. The items to be exported are the system gates `dqgetsize` and `putdescriptor` to the file named `system.sys` in module `sysgates`, gates `dqwrite` and `dqread`, and the public symbol identifying a global data area called `globaldata` to the file named `io.sys` in module `ioinfo`.

In the third exportation, two gates, `gate1` and `gate2`, are placed in module `xgates`. An entire module, `mod1`, is placed in module `xmod`. Two segments, `seg1` and `seg2`, are placed in module `xsegs`. Two descriptors, `globdata1` and `globdata2`, are placed in module `xdata`. All four modules become part of file `multi.sys`.

```
EXPORT
#system.sys  (sysgates  (dqgetsize,      --two gates are exported
                  putdescriptor)), -- to a module called
                                     -- sysgates in a file
                                     -- called system.sys

#io.sys      (ioinfo   (dqwrite,        --two gates and a pointer
                  dqread,             -- to a descriptor
                                     -- installed in
                  globaldata)), -- the GDT are placed in
                                     -- module ioinfo in a file
                                     -- called io.sys

#multi.sys   (xgates    (gate1,          --two gates in module
                  gate2),             -- xgates,
                  xmod      (mod1),    -- a module in xmod,
                  xsegs     (seg1, seg2), -- two segments in xsegs,
                  xdata     (globdata1, -- and pointers to two GDT
                  globdata2)); -- descriptors in xdata,
                                     -- are put in a file
                                     -- called multi.sys
```

Purpose and Scope

This definition defines call gates, task gates, interrupt gates, or trap gates. Two classes of gates are defined: 80286 gates with 80286 type-assignments, format, and semantics; and 80386 gates with new types, extended format, and different semantics. (There are no gates for 8086 code.) Only 80386 gates are allowed on the 80376. The segment that contains the entry point for a call, trap, or interrupt gate must be present in an input module. If a task gate points to a TSS, that TSS must be defined earlier in the build program with a TASK definition (see the TASK definition in this chapter).

Syntax

gate-definition ⇒ GATE *newgate* [, ...]

$$\text{newgate} \Rightarrow \text{gate-id} \left[\left(\left\{ \begin{array}{l} \text{ENTRY} = \text{entry-point} \\ \text{WC} = \text{word-count} \\ \text{DPL} = \text{priv-level} \\ \text{gate-type} \\ \text{presence-indicator} \\ \text{use-attribute} \end{array} \right\} \right) [, \dots] \right]$$

entry-point ⇒ { *public-id* | *task-id* }

word-count ⇒ { 0 | 1 | .. | 31 }

priv-level ⇒ { 0 | 1 | 2 | 3 }

gate-type ⇒ { CALL | INTERRUPT | TRAP | TASK }

presence-indicator ⇒ [NOT] PRESENT

use-attribute ⇒ { USE16 | USE32 }

GATE Definition (continued)

Where:

<i>gate-id</i>	is an identifier that names a gate. By default, BLD386 assumes that the <i>entry-point</i> has the same name as the <i>gate-id</i> .
<i>public-id</i>	is the entry point (the identifier for a public procedure in a code segment) for a CALL, TRAP, or INTERRUPT gate.
<i>task-id</i>	is the entry point (the identifier for a TSS) for a TASK gate. This name must already have been assigned in a TASK definition earlier in the build program.

Discussion

By default, the *entry-point* is the same as the *gate-id*. The *priv-level* (descriptor privilege level) is 3. The *presence-indicator* (present bit) is 1, PRESENT, indicating that the segment is in memory. The *word-count* is 0 unless a word count is specified in the public procedure identified with *entry-point*. The *use-attribute* is USE32. The *gate-type* is a CALL gate.

The *word-count* is valid only for call gates. It establishes the number of words to pass from the caller's stack to the target procedure's stack. Up to 31 words can be passed.

The *priv-level* defines the gate's privilege level. It is a number from 0 through 3, with 0 representing the highest privilege level.

The *gate-type* defines the type of gate being created: CALL, TASK, INTERRUPT, or TRAP. The CALL, TRAP, or INTERRUPT types are either 80286-type (USE16) or 80386-type (USE32) to correspond with the D-bit in the descriptor of the code segment of *entry-point*. The Type bits are 1100 for an 80386 CALL gate, 0101 for a TASK gate, 1110 for an INTERRUPT gate, or 1111 for an 80386 TRAP gate. The *gate-type* is compared to the type of code segment being addressed (e.g., an 80386 CALL gate must address an 80386 code segment). Only USE32 gates can be used in an object module for an 80376 target.

GATE Definition (continued)

The *presence-indicator* controls whether the P-bit in the gate descriptor is 1, PRESENT (indicating that the segment is in memory), or 0, NOT PRESENT.

The *use-attribute* (which is the D-bit in the Type bits of the gate descriptor) can override the default, USE32.

The fields in a gate descriptor are shown in Figure 3-2.

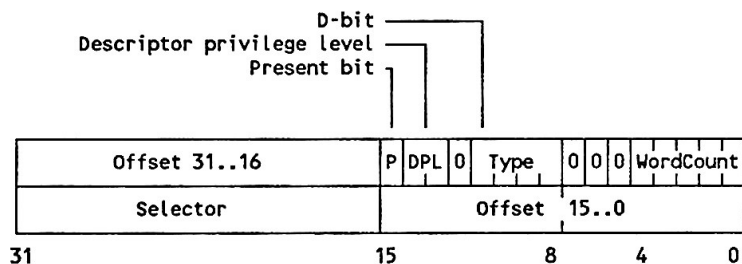


Figure 3-2 Gate Descriptor Format

GATE Definition (continued)

Table 3-3 defines the relationship between the *gate-definition* and the fields in a gate descriptor.

Table 3-3 Syntax Summary for Gate Definition

Descriptor Field	Definition Parameter	Values	Default
Selector and Offset	<i>entry-point</i>	ENTRY= <i>public-id</i>	**
WordCount	<i>word-count</i>	WC={0 1 ...31}	*
Type is 1100b (call gate)	<i>gate-type</i>	CALL	CALL
	<i>use-attribute</i>	{USE16 USE32}	USE32
Selector and Offset	<i>entry-point</i>	ENTRY= <i>task-id</i>	**
Type is 0101b (task gate)	<i>gate-type</i>	TASK	-
Selector and Offset	<i>entry-point</i>	ENTRY= <i>public-id</i>	<i>gate-id</i>
Type is 1110b or type is 1111b (interrupt or trap gate)	<i>gate-type</i>	INTERRUPT TRAP	-
	<i>use-attribute</i>	{USE16 USE32}	USE32
DPL	<i>priv-level</i>	DPL={0 1 2 3}	DPL=3
P	<i>presence-indicator</i>	[NOT] PRESENT	PRESENT

Notes:

*BLD386 assigns a value of 0, unless a word count is specified in the public procedure identified in entry-point.

**The default entry point is the public procedure or the task with the same name as gate-id.

NOTES

Ensure that the gate's *priv-level* is numerically greater than or equal to that of the target.

There is no difference between an 80286 and an 80386 TASK gate, and BLD386 issues a warning if the *use-attribute* is specified for a TASK gate.

8086 code is not addressed by gates; therefore, USERREAL is not an available attribute for a gate.

BLD386 issues an error message if the USE16 attribute is specified for a gate when the MOD376 control is in effect (see Chapter 4).

Ensure that the type of gate being created is compatible with the target containing the *entry-point*. For example, a TASK gate must point to a TSS, and a CALL gate must point to a code segment.

Examples

1. This example shows a definition of a TASK gate and an INTERRUPT gate. The first gate points to the task block named `task1`, and the second gate points to the interrupt procedure `errhandler`. Both gates have privilege level 3 by default.

GATE

```
taskentry      (TASK,          --a task gate at DPL=3
               ENTRY = task1), -- points to TSS task1

dqerrorhandler (INTERRUPT,     --interrupt gate points
               ENTRY = errhandler); -- to errhandler
```

GATE Definition (continued)

2. This example defines three gates, all CALL gates by default. The first gate, `dqgetsize`, points to public procedure `getsize`, and has a privilege level of 3 by default. Its word count is taken from the public definition for symbol `getsize` in an input module. The next gate, `dqwrite`, is a privilege-level 2 gate pointing to the public procedure `xqwrite`. The last gate, `abort`, is a default privilege-level 3 gate pointing to a public procedure with the same name, and its word count is taken from the public definition of the symbol `abort` in the input module.

GATE

```

dqgetsize      (ENTRY = getsize),  --a call gate at DPL=3
                                     -- points to public
                                     -- procedure getsize

dqwrite        (ENTRY = xqwrite,    --a call gate at DPL=2
                 DPL = 2, WC = 4),  -- passes 4 words to
                                     -- procedure xqwrite

abort;          --a call gate at DPL=3
                 -- points to public
                 -- procedure abort
```

Purpose and Scope

This definition specifies the layout of memory. The memory layout can be specified by reserving specific memory locations, and by defining specific memory ranges to be allocated for designated entities or classes of entities. Each named memory range must be qualified as RAM or ROM memory. This definition is ignored if the NOBOOTLOAD control is included in the invocation (see Chapter 4).

Syntax

```
memory-definition ⇒ MEMORY mem [ ,... ]

mem ⇒ ( setup [ ,... ] )

setup ⇒ { [ RESERVE = ( reserve-def [ ,... ] ) ]
          [ RANGE   = ( range-def   [ ,... ] ) ]
          [ ALLOCATE = ( alloc-def   [ ,... ] ) ] }

reserve-def ⇒ number1 .. number2 [ PAGED ]

range-def ⇒ range-id = { RAM | ROM } ( range-list )

range-list ⇒ { number3 .. number4 } [ ,... ]

alloc-def ⇒ range-id = ( name [ ,... ] )

name ⇒ { identifier | *TABLES | *SEGMENTS | *TASKS }
```

Where:

number1..number2 must be defined with 32-bit addresses with *number2* greater than *number1*.

MEMORY Definition (continued)

<i>range-id</i>	is the name assigned to the memory locations between the absolute addresses <i>number3</i> and <i>number4</i> , inclusive. Multiple memory areas can be included in a single <i>range-id</i> . The keyword RAM or ROM must be specified. This keyword indicates that the specified memory address space has either read/write or read-only access, respectively.
<i>number3..number4</i>	must be defined with 32-bit addresses with <i>number4</i> greater than <i>number3</i> .
<i>name</i>	can be the <i>identifier</i> of a specific table, segment, or task; *SEGMENTS, *TABLES, or *TASKS. Each of these three predefined keywords designates a complete collection of such entities, e.g., *SEGMENTS indicates all segments, etc.

Discussion

Use the RESERVE construct to specify the memory blocks that are to be excluded from memory allocation by BLD386. The memory areas between *number1* and *number2* are reserved and therefore not available for program segments, TSSs, or tables.

The PAGED attribute specifies that the memory area defined by *number1* and *number2* is included in page tables. It is valid only for reserved memory areas. PAGED is effective only if PAGETABLES, MOD386, and BOOTLOAD controls are also in effect (see Chapter 4 for information on controls). The PAGING definition described in this chapter is another way to specify memory to be included in page tables.

Use RANGE to define one or more memory blocks of contiguous bytes in memory. Each such memory range is associated with a user-defined name, *range-id*. One of the keywords, RAM or ROM, must be specified in the range definition. The range names are used in the ALLOCATE construct, and possibly in the PAGING definition later in the build program.

MEMORY Definition (continued)

Use the **ALLOCATE** construct to place one or more items in the predefined ranges. The **RANGE** construct must be used before the **ALLOCATE** construct because a *range-id* must be defined before it can be used.

When the keywords ***TABLES**, ***SEGMENTS**, or ***TASKS** are used in the **ALLOCATE** construct, only entities which have been declared before the *memory-definition* are included. Exceptions to this are the default GDT and IDT. (BLD386 does no forward-referencing; for entities declared after the *memory-definition* in the build program, BLD386 assigns base addresses within the specified range only if the base specified in the respective build program definition is within that range.)

Input items that are not specified in any **ALLOCATE** construct are allocated space in the specified ranges according to their access attributes (e.g., read-only, execute-only, and execute/read segments go to ROM; read/write segments go to RAM).

BLD386 follows this sequence when allocating memory:

1. Process the *reserve-def* (if any).
2. Assign addresses to entities for which a base address has been specified in a build program definition. If memory is to be overlaid, a base address must be specified in the appropriate build program definition.
3. If **RANGE** or **ALLOCATE** constructs are used, BLD386 assigns addresses for entities accordingly.

Entities that are not explicitly allocated with a *range-def* are placed in a *range-def* memory area which has the ROM attribute.

4. If **RANGE** or **ALLOCATE** constructs are not used, BLD386 assigns addresses to tables, then tasks, then segments that do not have a base address specified in a build program definition.
5. All alias segments are assigned bases.

Addresses within a segment can wrap from high memory to low memory. BLD386 adjusts default address assignments accordingly and does not overlay low addresses.

NOTES

Mismatches can occur when using an ALIAS definition and the *memory-definition* in the same build program. If a segment that is also an alias is specified in the RANGE or ALLOCATE construct, then that segment is allocated according to the RANGE construct. The alias segment allocation has lower priority. In this case, BLD386 issues a warning.

Because BLD386 processes the build program definitions in a single pass and does no forward-referencing, all information needed to comply with a definition must be available to BLD386 when it processes that definition. If a CREATESEG definition, a TABLE definition with an LDT, or a TASK definition is processed after the ALLOCATE construct, BLD386 does not add these new entities to the ranges specified, nor does it process them with regard to the ALLOCATE specification.

Arrange the *memory-definition* constructs in the order shown in the syntax definition, thus: RESERVE, RANGE, ALLOCATE.

The *memory-definition* is ignored if the NOBOOTLOAD control is included in the invocation (see Chapter 4 for details on NOBOOTLOAD).

Example

This example specifies the memory layout to be used by BLD386 in its allocation scheme. Figure 3-3 shows the resulting layout.

MEMORY

```
(RESERVE = (100..200, 300..500),          --RESERVE 2 memory areas

RANGE    = (local_mem = ROM (1000..2000,   --define 3 RANGE names
              4000..6000),
              shared_mem = RAM (10000..20000),
              read_only  = ROM (8000..9000))

ALLOCATE = (local_mem = (*TABLES, *TASKS), --ALLOCATE entities in the
              shared_mem = (code32, data,   -- specified RANGES
                            common_seg),
              read_only  = (const_seg))

);
```

MEMORY Definition (continued)

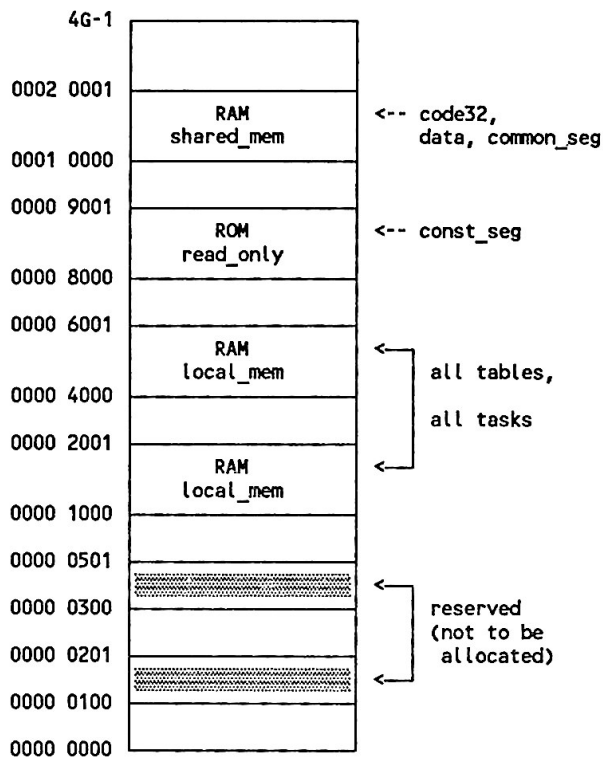


Figure 3-3 Example Memory Layout

Purpose and Scope

This definition defines one directory and any number of page tables for bootloadable output if the PAGETABLES control is specified in the invocation (see Chapter 4 for information on controls). This definition is invalid if the MOD376 control is in effect. The user-defined (UD1, UD2, and UD3), user/supervisor (US), read/write (RW), or present (P), bits for any individual page or page table can also be specified.

Syntax

paging-definition \Rightarrow *PAGING makepages*

$$\text{makepages} \Rightarrow \text{dirseg-id} \left(\left\{ \begin{array}{l} \text{maketables} \\ \text{bits} \\ \text{LOCATION} = \text{public-id} \end{array} \right\} [, \dots] \right)$$

maketables \Rightarrow *PAGETABLES page-seg-id* [(*paging-list*)]

$$\text{paging-list} \Rightarrow \left\{ \begin{array}{l} \text{range1-id} \\ \text{number1} \dots \text{number2} \end{array} \right\} [, \dots]$$

bits \Rightarrow *BITSETTING* (*range-action* [, ...])

$$\text{range-action} \Rightarrow (\text{set-bit} [, \dots]) \left\{ \begin{array}{l} \text{range2-id} \\ \text{number3} \dots \text{number4} \end{array} \right\}$$

set-bit \Rightarrow *bit-action bit-name*

bit-action \Rightarrow { SET | RESET | DSET | DRESET }

bit-name \Rightarrow { P | RW | US | UD1 | UD2 | UD3 }

PAGING Definition (continued)

Where:

dirseg-id is the name of the segment that holds the page directory. If the segment already exists, it must have the following attributes: readable, writable, page alignment, and not expanddown (see SEGMENT definition in this chapter). The starting point of the directory is the current size of the segment, rounded up to the next page boundary.

If the segment is new, the directory starts at the first byte of the new segment, and the limit of the segment is extended by 4K bytes to allow space for the directory. The following attributes are assigned (see SEGMENT definition in this chapter): readable, writable, a privilege level of 0, a limit of 4095 (4K bytes-1), and page alignment. A descriptor for the segment is placed in the GDT.

page-seg-id is the name of the segment which holds page tables. The *page-seg-id* can be an existing segment (including the directory segment), or it can be a new segment. The same rules apply for *page-seg-id* segments as given above for *dirseg-id* segments; however, the size of the new *page-seg-id* segment, or extension of an existing segment, is 4K bytes for every 4M bytes (or part thereof) of memory included to be paged.

public-id is a public symbol where BLD386 can store the real address of the page directory. This value must be the address of a memory buffer at least four bytes long in a read/write, non-executable segment.

range1-id is a name which has already been assigned to a range of memory locations in the MEMORY definition.

number1..number2 must be defined with 32-bit addresses with *number2* greater than *number 1*.

PAGING Definition (continued)

range2-id is a name which has already been assigned to a range of memory locations in the MEMORY definition.

number3..number4 must be defined with 32-bit addresses with *number4* greater than *number3*.

Discussion

Figure 3-4 shows the page directory and page table entry formats.

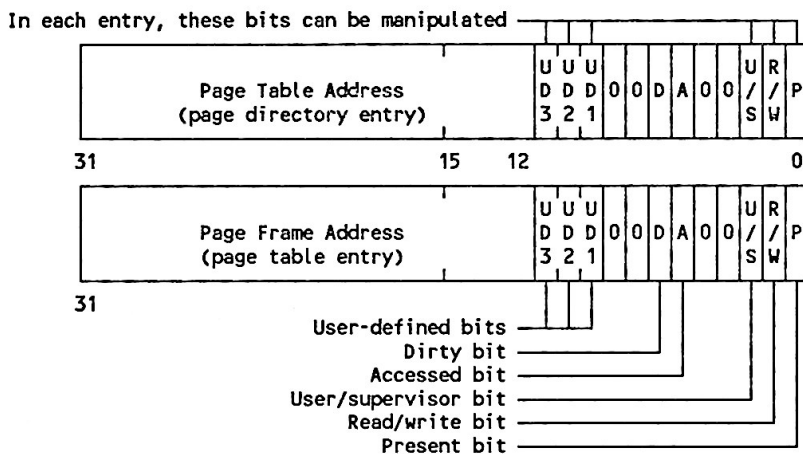


Figure 3-4 Page Directory and Page Table Entries

The *paging-list* is a list of memory ranges to be included in the page tables. This list can contain *range1-ids* which were defined in the MEMORY definition section or explicit ranges denoted *number1..number2*. Note that the *paging-list* must refer to complete pages; i.e., the lower address must be page-aligned and the upper address must be page-aligned minus 1. Ranges must not overlap. If no *paging-list* is given, then the default memory to be included in the page tables is 4G bytes minus the memory defined as reserved in the MEMORY definition section. Paged memory must be defined either

PAGING Definition (continued)

explicitly with the *paging-definition*, or implicitly using the RESERVE and PAGED constructs in the MEMORY definition.

If the PAGETABLES option is not specified in the *paging-definition*, then BLD386 builds page tables and a directory for all memory defined in the MEMORY definition as being paged. The page tables are placed in the same segment as the page directory.

BITSETTING controls flags for the directory entries or page table entries. BITSETTING can override defaults. *number3* and *number4* indicate 32-bit addresses.

SET and DSET indicate making a bit equal to 1. RESET and DRESET indicate a 0.

SET and RESET operate on the bit for the corresponding range's entry in the page table. The range must refer to complete pages.

DSET and DRESET operate on the bit for the corresponding range's entry in the page directory. The range must refer to complete 4M byte areas.

P is the present bit, which is 1 by default. RW is the read/write bit, which is also 1 by default. US is the privilege level 3 access or user/supervisor bit, also 1 by default. UD1, UD2, and UD3 are user-definable bits; all are 0 by default.

NOTES

The page directory address is also placed in every TSS (see TASK definition in this chapter).

Each page directory entry must map a contiguous 4M bytes of linear memory aligned on a 4M byte boundary. BLD386 maps linear to real memory directly. If the *paging-list* does not represent a multiple of 4M bytes, then BLD386 issues a warning and pads the page tables with entries representing pages "not present".

PAGING Definition (continued)

If the SEGMENT definition's SPECLN is greater than 0 in any segment (i.e., the pages within the segment must be locked in and are unswappable), BLD386 sets the UD1 bit to 1 in all page table entries pointing to that segment. Entries in the page directory pointing to those page tables also have their UD1 bits set to 1. Finally, any directory or page table entries pointing to the segment which contains the page directory itself also have their UD1 bits set to 1. Thus, not only are the actual pages of the segment locked in, but the directory and all related parts of page tables are also locked in. The actual method of indicating unswappable pages depends on the operating system. The choice of setting UD1 is arbitrary and provided for convenience.

Do not confuse the PAGETABLES construct in the *paging-definition* with the PAGETABLES control for invocation (see Chapter 4).

The *paging-definition* is invalid when the MOD376 control is in effect (see Chapter 4).

Examples

1. This example shows how to use the RESET construct in bit setting to prevent access to a part of memory by privilege-level 3 code.

BITSETTING	--partial example shows first 4M bytes
((DSET P, DSET RW, DSET US) 0..3FFFFFFh,	-- available, but does not allow
(RESET RW, RESET US) 0..1FFFFh)	-- level-3 access to first 8K bytes

2. This example shows two page tables created for 8M bytes of memory, one page directory, and both the tables and directory reside in a segment named pagedirseg.

MEMORY	--two page tables and one
(RESERVE = (800000h .. 0FFFFFFFh))	-- directory built for
	-- 8M bytes of memory,
PAGING pagedirseg;	--all pages have P, RW, and US
	-- set to 1

PAGING Definition (continued)

3. This example again creates page tables and a directory both in segment `pagedirseg`. The `table_area` containing the GDT and IDT is not accessible to privilege-level 3 code. The memory range `eprom1` is fixed, and not to be considered swappable. The UD1 bit is set to reflect this for use by this operating system.

MEMORY

```
(RESERVE = (500000h .. 0FFFFFFh),           --only 5M bytes exist
RANGE =   (eprom1 =   ROM(0..03FFFh),         -- including 12K of eprom
           ram1  =   RAM(5000h .. 3FFFFFh),    -- and RAM is remainder
           table_area = RAM(4000h .. 4FFFh)),
ALLOCATE = (table_area = (GDT, IDT))
);
```

```
PAGING pagedirseg           --this is page directory location,
(PAGETABLES pagedirseg      --the tables are in same
                             -- segment as directory,
                             (eprom1, ram1, ram2, table_area), --these are ranges to be paged
```

BITSETTING

```
((SET UD1) eprom1,          --the o.s. can use UD1 as
                             -- fixed page flag, and
                             (RESET US) table_area) -- disallow level-3 access
);                             -- to table area
```


Purpose and Scope

This definition defines individual fields in descriptors for segments in the input modules. Assign values to individual segment descriptors, or to descriptors for all segments in an input module.

Syntax

segment-definition \Rightarrow SEGMENT *seg* [, ...]

$$seg \Rightarrow seg\text{-}name \left(\left\{ \begin{array}{ll} BASE & = base\text{-}addr \\ LIMIT & = segment\text{-}limit \\ ALIGN & = align\text{-}type \\ SPECLEN & = number32 \\ DPL & = priv\text{-}level \\ seg\text{-}attribute \\ use\text{-}attribute \end{array} \right\} [, \dots] \right)$$

$$seg\text{-}name \Rightarrow \left\{ \begin{array}{l} module\text{-}id \\ [module\text{-}id.] combine\text{-}id \\ *SEGMENTS \end{array} \right\}$$

base-addr \Rightarrow { *number32* | AT (*public-id*) }

segment-limit \Rightarrow [+ | -] *number32*

$$align\text{-}type \Rightarrow \left\{ \begin{array}{l} BYTE \\ WORD \\ DWORD \\ QWORD \\ PARAGRAPH \\ INPAGE \\ PAGE \end{array} \right\}$$

priv-level \Rightarrow { 0 | 1 | 2 | 3 }

SEGMENT Definition (continued)

$$seg_attribute \Rightarrow \left\{ \begin{array}{l} [NOT] \text{ WRITABLE} \\ [NOT] \text{ EXPANDDOWN} \\ [NOT] \text{ READABLE} \\ [NOT] \text{ PRESENT} \\ [NOT] \text{ CONFORMING} \end{array} \right\}$$

use-attribute \Rightarrow { USE16 | USE32 | USEREAL }

Where:

- seg-name* identifies the segment. Segments may be specified by module name, either individually or as a combination of two names, or by the predefined keyword *SEGMENTS (which indicates all segments in the input modules).
- module-id* is an identifier that refers to one of the linkable input modules, and thereby to all the segments in that module unless the *.combine-id* suffix is attached. If only a *module-id* is specified, the values specified in all other parameters are applied to all segment descriptors for the input module named *module-id*.
- combine-id* is the combine name of a segment in one of the linkable input modules. The *combine-id* refers to a specific segment. When there are multiple input segments with the same name, a *module-id* should be specified with a *combine-id* to identify a specific instance of the segment. (Remember that BLD386 does not combine segments between modules.)
- public-id* is a symbol that defines the base address of the segment. Use the symbolic definition to create overlapping segments. The *public-id* must be a public identifier and belong to a segment that has already been assigned its absolute address. The base of the segment identified by *seg-name*, is set to the address of the *public-id*.

Discussion

Figure 3-5 shows the fields in a segment descriptor.

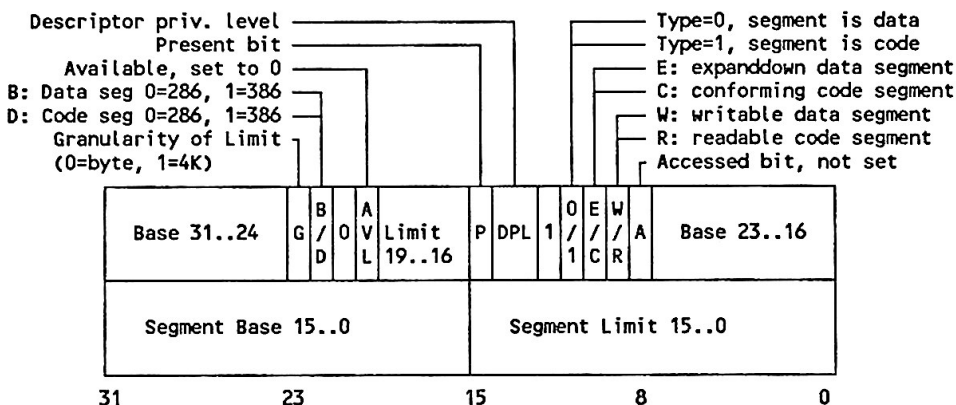


Figure 3-5 Segment Descriptor Format

By default, the corresponding segment definition records in the input modules provide the segment limits, privilege levels, and attributes. The base address can be specified explicitly or symbolically, and the present bit is 1, PRESENT. Specification of a base address overrides the default memory allocation scheme.

Less-privileged code can access entry points in a code segment with the CONFORMING attribute without going through a gate (see the GATE definition in this chapter).

BLD386 does not combine input segments except within the same module to create DSC (data/stack combined) segments from data and stacks in small-model programs.

The LIMIT construct sets or modifies the number of bytes that constitute the segment length. This value, equal to the segment length minus one, can be set to a specific byte value (*number32*), increased by a number of bytes (*+number32*), or decreased by a number of bytes (*-number32*).

SEGMENT Definition (continued)

Granularity characterizes the segment limit. If the limit is greater than 16M bytes, it is rounded up to a multiple of 4K bytes, then divided by 4K bytes, and the number of 4K-byte pages is stored in the limit field of the descriptor. The G-bit is set to 1 to show that the limit field is a multiple of 4K bytes. The G-bit has no effect on the BASE construct, since the base address is given as a full 32-bit byte-granular address. When G=0 (80286 compatible), the limit is calculated by extending the limit field to 32 bits with high-order zeros. When G=1, the limit is calculated by shifting the limit field left 12 bits with low-order ones inserted.

For data/stack combined (DSC) and other expanddown segments in which the limit (in the descriptor) is the complement of the actual size, the size appears as a high number. For expanddown segments, the limit is rounded up to make the effective limit a multiple of one page. Figure 3-6 shows the organization of a data/stack combined segment.

BLD386 checks the B/D-bit in the segment descriptor for base and limit constraints. D stands for "default operand size" for code segments. B stands for "big", meaning the size (big or small) for data segments. The language translator sets the B/D-bit, but it can be set or cleared by specifying a USE32 or USE16 *use-attribute*. The B/D-bit indicates whether the segment is an 80286- or 80386-type. If the bit is 1, then the segment is an 80386-type, and its base and limit can be 32-bit numbers (up to 4G bytes-1). The minimum size for big expanddown stacks is 4K bytes. If the bit is 0, then the segment is 80286-type, and its base cannot be more than a 24-bit number (up to 16M bytes-1), and its limit cannot be more than a 16-bit number (up to 64K bytes-1). If the MOD376 control is in effect, the *use-attribute* can only be USE32 (see Chapter 4 for information on the MOD376 control).

USEREAL indicates an 8086-type code segment. 8086 instructions assume that the D-bit is off. This kind of segment does not have a descriptor. Addressing such a segment is done by paragraph number rather than descriptor number. A USEREAL segment must be paragraph-aligned either explicitly with the ALIGN or BASE constructs or defined as such in the input object file. USEREAL is effective only in bootloadable files (see the BOOTLOAD control in Chapter 4). USEREAL is not available on the 80376.

SEGMENT Definition (continued)

The *use-attribute* of a DSC segment is USE16 if both its constituent segments have the USE16 attribute, or USE32 if both its constituents have the USE32 attribute. If one segment is USE16 and the other is USE32, the DSC is USE16.

BLD386 can add extra bytes to segments in addition to the gaps introduced for alignment. This length adjustment is referred to as padding. BLD386 pads a USE16 segment having read-only (RO), read/write (RW), or execute/read (ER) access rights, so that 16-bit word references to the last byte in the segment remain valid. BLD386 pads a USE32 segment having RO, RW, or ER access rights, so that word-long references to the last byte in the segment remain valid. When adding bytes to a data/stack combined (DSC) segment, BLD386 adds bytes to the data portion of the segment; the stack size remains unchanged.

The size of the data portion of a USE16 DSC segment is even, and the size of the data portion of a USE32 DSC segment is divisible by four. If adding the byte or bytes makes the data portion of a USE16 DSC segment an odd length, or makes the data portion of a USE32 DSC segment not divisible by four, BLD386 pads the data portion of the segment with one or more bytes. Note that because the length of the data portion of a DSC segment is even or divisible by four, its effective stack pointer is also even or divisible by four.

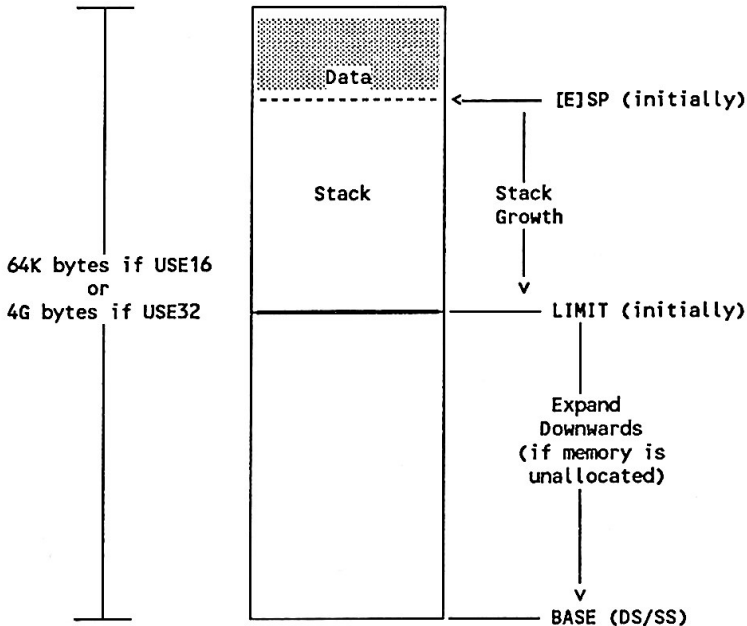


Figure 3-6 Data/Stack Combined Segment Organization

The **ALIGN** parameter controls alignment of the segment. A segment may be aligned to a **BYTE**, **WORD**, **DWORD**, **QWORD**, **PARAGRAPH**, **INPAGE**, or **PAGE** boundary. **INPAGE** alignment is especially important for TSSs, which are paragraph-aligned but must not cross a page boundary. Input object files that come from a language translator have their alignment set by the language processor. For entities created by BLD386, TSSs are paragraph-aligned, descriptor tables are word-aligned, page tables are page-aligned, an 80386-type stack or DSC segment is dword-aligned, and an 80286-type stack or DSC segment is word-aligned.

The **SPECLN** construct defines how much of a segment must not be paged out by the paging mechanism (by setting the **UDI** bit of the page or pages to 1 in corresponding page tables). See the **PAGING** definition in this chapter. **SPECLN** is the number of bytes starting from offset 0 in the segment. Those parts of code that must respond

SEGMENT Definition (continued)

quickly and thus never be paged out are "locked in" by specifying a SPECLEN large enough to cover the critical code. For example, the code of a disk driver would have its pages "locked in". The SPECLEN of a segment may not be larger than its size. If a larger SPECLEN is specified, it is truncated to the segment's limit, plus one with no warning. The segment is also PAGE aligned.

The DPL is the descriptor privilege level. The DPL is a number from 0 through 3, where 0 signifies the highest privilege. BLD386 is the only utility that can build a system with multiple privilege levels.

Table 3-4 shows a summary of the syntax used to define a segment.

SEGMENT Definition (continued)

Table 3-4 Syntax Summary for Segment Definition

Descriptor Field	Definition Parameter	Values	Default
Base	<i>base-addr</i>	$\text{BASE} = \{\text{number32} \mid \text{AT } (\text{public-id})\}$	**
Limit	<i>segment-limit</i>	$\text{LIMIT} = [+ \mid -] \text{number32}$	***
	<i>align-type</i>	$\text{ALIGN} = \{\text{BYTE} \mid \text{WORD} \mid \text{DWORD} \mid \text{QWORD} \mid \text{PARAGRAPH} \mid \text{PAGE} \mid \text{INPAGE}\}$	*
		$\text{SPECLEN} = \text{number32}$	***
DPL	<i>priv-level</i>	$\text{DPL} = \{0 \mid 1 \mid 2 \mid 3\}$	*
Type is 0 data segment	<i>use-attribute</i>	$\{\text{USE16} \mid \text{USE32}\}$	**
P	<i>seg-attribute</i>	[NOT] PRESENT	*
E		[NOT] EXPANDDOWN	*
W		[NOT] WRITABLE	*
Type is 1 code segment	<i>use-attribute</i>	$\{\text{USE16} \mid \text{USE32} \mid \text{USEREAL}\}$ ****	**
P	<i>seg-attribute</i>	[NOT] PRESENT	*
C		[NOT] CONFORMING	*
R		[NOT] READABLE	*

Notes:

*Extracted from input segments.

**BLD386 assigns a value to this field if the BOOTLOAD control is in effect (see Chapter 4).

***Assigned by BLD386.

****USEREAL 8086-type segments do not have descriptors, but are addressed by paragraph number.

SEGMENT Definition (continued)

NOTES

BLD386 issues a warning if the specified segment or module does not exist.

The default values of parameters of a segment can be overridden in several different *segment-definitions*. For each parameter, BLD386 takes the last value specified among the different definitions. Avoid using a *segment-definition* for the same segment more than once. A valid combination of parameters can be rendered invalid by subsequent *segment-definitions*.

The *use-attribute* defines the B/D-bit in a segment descriptor. 80286 translators always clear the B/D-bit to 0 (i.e., USE16), and 80386 translators always set the B/D-bit to 1 (i.e., USE32). The D-bit affects many 80386 instructions, and the B-bit affects stack operations. Avoid changing the B/D-bit.

If the MOD376 control is in effect, and a USE16 or USERREAL attribute is specified, BLD386 issues an error.

No descriptor is generated for USERREAL segments.

If the ALIGN option conflicts with the base address (e.g., a word-aligned segment is on an odd address), BLD386 issues a warning and ignores the alignment.

SEGMENT Definition (continued)

Examples

1. This example sets the DPL of all segments in module `osmod` to privilege level 0. All executable segments in module `mathmod` are conforming (not requiring gates for entry). The base of the code segment is set explicitly, and the limit of the stack segment is increased by 100 bytes. The limits of the other segments in `mathmod` are taken from the input module.

```
SEGMENT                                --all segments in module osmod
                                        -- are assigned DPL of 0, osmod (DPL = 0),

mathmod (CONFORMING),                 --all executable segments in module
                                        -- mathmod are made conforming,

osmod.code (BASE = 1000h),             --segment code in module osmod
                                        -- starts at 1000h,

stack (LIMIT = +100);                 --the stack size is increased by
                                        -- 100 bytes
```

2. This example sets the privilege levels for three segments in the module `sample`. It converts the `seg0` segment to an expanddown (stack) segment. The data segment `massive.data` has a 20-bit limit, and must be defined as `USE32`.

```
SEGMENT

DPL = 0),                             -- is downward expandable
                                        -- with a DPL of 0,

sample.seg1 (DPL = 1),                 --the DPL of seg1 in module sample
                                        -- is 1,

sample.seg2 (DPL = 2),                 --the DPL of seg2 in module sample
                                        -- is 2,

massive.data (LIMIT = 0FFFFh,         --the segment data in module massive
              USE32);                  -- has a limit of 0FFFFh and be 32-bit
```

Purpose and Scope

This definition creates descriptor tables: a GDT, an IDT, or LDTs. It also installs segment and system descriptors in the tables and reserves table slots. Use the TABLE definition only to install descriptors for segments found in input files, TSSs, LDT descriptors, and gates defined earlier in the build program.

For bootloadable output, BLD386 creates an alias for each descriptor table by default. This alias describes a segment with read/write attributes. For loadable output, BLD386 creates alias descriptors for LDTs only.

Syntax

table-definition ⇒ TABLE *newtable* [, ...]

$$newtable \Rightarrow table-name \left(\left\{ \begin{array}{ll} BASE & = base-addr \\ LIMIT & = table-lim \\ DPL & = priv-level \\ ENTRY & = (entry-list) \\ RESERVE & = (res-list) \\ LOCATION & = buf-id \\ presence-indicator & \\ output-indicator & \end{array} \right\} [, \dots] \right)$$

table-name ⇒ { GDT | IDT | *ldt-id* }

base-addr ⇒ { *number32* | AT (*public-id*) }

table-lim ⇒ [+] *slots*

slots ⇒ { 0 | 1 | .. | 8190 }

priv-level ⇒ { 0 | 1 | 2 | 3 }

$$entry-list \Rightarrow \left\{ \begin{array}{l} [index :] entry \\ index : [(entry [, \dots])] \end{array} \right\} [, \dots]$$

TABLE Definition (continued)

index \Rightarrow { 0 | 1 | .. | 8190 }

entry \Rightarrow { *seg-name* | *gate-id* | *task-id*[.LDT] | *anldt-id* }

seg-name \Rightarrow $\left\{ \begin{array}{l} \text{module-id} \\ [\text{module-id.}] \text{combine-id} \end{array} \right\}$

res-list \Rightarrow { *index1* .. *index2* } [, ...]

index1 \Rightarrow { 0 | 1 | .. | 8190 }

index2 \Rightarrow { 0 | 1 | .. | 8190 }

presence-indicator \Rightarrow [NOT] PRESENT

output-indicator \Rightarrow [NOT] CREATED

Where:

ldt-id is the identifier of an LDT. A public definition is created for *ldt-id*. BLD386 creates an LDT (named LDT?) automatically when uninstalled segment-descriptors, call gates, or task gates exist. To create an LDT, provide the *ldt-id* to be associated with the LDT and the list of descriptors to be installed in the LDT.

public-id defines the *base-addr* of the table. The *public-id* must be a public identifier and belong to a segment that has already been assigned its absolute address. The base of the table identified by *table-name* is set to the address of the *public-id*.

slots defines the total number of descriptor slots in a table. Valid ranges are as follows:

- 1 through 8190 for the GDT
- 0 through 255 for the IDT
- 0 through 8190 for an LDT

Specifying *+slots* adds the specified number of slots to those already defined explicitly or by default for a descriptor table.

TABLE Definition (continued)

index

represents an index position in the table. Valid ranges are as follows:

- 1 through 8190 for the GDT
- 0 through 255 for the IDT
- 0 through 8190 for an LDT

BLD386 assigns indexes sequentially in each table, accommodating any user-defined *index* values and any reserved slots. If an *index* is specified without an *entry*, the slot is filled with zeros.

entry

is an identifier for a segment, gate, task, or LDT for which a descriptor is placed in the table. The identifier specified must have been defined earlier in the build program or must exist in an input module. Table 3-5 shows the identifiers which are valid *entry* names for the table type indicated.

Table 3-5 Valid Entries for Tables

Identifier for Entry	GDT	IDT	LDT
<i>seg-name</i>	x	x	
<i>gate-id</i>	call or task gate	interrupt or trap gate	call or task gate
<i>task-id</i> [.LDT]	x		
<i>anldt-id</i>	x		

If one *index* is specified before a list of *entries*, the *index* slot is assigned to the first name specified in that list. The rest of the *entries* in the list are assigned sequential slots following the *index* value specified.

TABLE Definition (continued)

<i>module-id</i>	refers to all segments in a module, unless used in combination with <i>combine-id</i> . When only <i>module-id</i> is specified, BLD386 installs the descriptors for all the segments in the same order as the segments appear in the module.
<i>combine-id</i>	identifies a specific segment when more than one input segment may have the same name.
<i>gate-id</i>	is the gate name assigned in a GATE definition.
<i>task-id</i>	is the task name assigned in a TASK definition. The <i>task-id</i> is TASK? for an automatically created TSS (see TASK definition in this chapter). A <i>task-id</i> .LDT refers to the LDT for the task named by <i>task-id</i> .
<i>anldt-id</i>	is the name of an LDT assigned in a TABLE definition.
<i>buf-id</i>	is a public symbol attached to a buffer where BLD386 can store the <i>base-addr</i> and limit of the newly-defined table. BLD386 places the <i>base-addr</i> and the limit of the table being defined in a buffer starting at <i>buf-id</i> . This buffer must be at least 6 bytes long. The base and limit values are placed in this buffer as two bytes of limit and four bytes of base in the format required by the ASM386 LGDT and LIDT instructions (load GDT register and load IDT register).

Discussion

For bootloadable modules, BLD386 creates a default IDT that contains 32 slots, plus any additional specified entries. The default slots, numbered 0-31, correspond to Intel-defined or Intel-reserved interrupts. To override this default, specify a larger or smaller number of slots in the LIMIT construct.

For bootloadable modules, BLD386 automatically creates two alias descriptors for table segments, one for the GDT, and one for the IDT. The syntax summary for the table aliases is shown in Table 3-6. These descriptors have privilege level 0 and a *presence-indicator* of 1. They are installed in the GDT slots 1 and 2 as shown in Figure 3-7.

TABLE Definition (continued)

Table 3-6 Syntax Summary for Alias Descriptor for Table Segment

Descriptor Field	Definition Parameter	Values	Default
Base	<i>base-addr</i>	BASE = { <i>number32</i> AT (<i>public-id</i>)}	*
Limit	<i>table-lim</i>	LIMIT = [+] slots	**
DPL	<i>priv-level</i>	DPL = {0 1 2 3}	DPL = 0
P	<i>presence-indicator</i>	[NOT] PRESENT	PRESENT
	<i>output-indicator</i>	[NOT] CREATED	CREATED

Notes:

*BLD386 assigns a value to this field if the BOOTLOAD control is in effect. See Chapter 4.

**The value of *table-lim* is table dependent. For the GDT, *table-lim* is in the range 1..8190. For the IDT, *table-lim* is in the range 0..255. For LDTs, *table-lim* is in the range 0..8190. The *table-lim* by default is set to the number of descriptors installed in the table. Limit is computed as follows: (8 bytes per slot * number of slots) - 1.

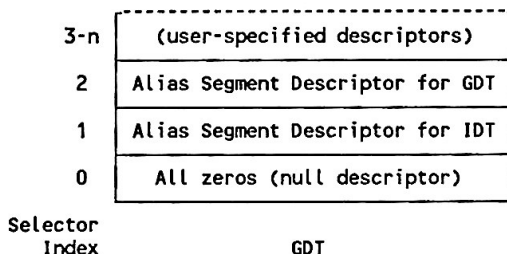


Figure 3-7 Default Table Organization for GDT

BLD386 creates an LDT named LDT? automatically when uninstalled segment descriptors, call gates, or task gates exist. The default LDT organization is shown in Figure 3-8.

2-n	(user-specified descriptors)
1	Alias Segment Descriptor for LDT
0	All zeros (null descriptor)

Selector Index LDT

Figure 3-8 Default Table Organization for LDT

The *priv-level* defines the privilege level of the table being defined. It may be any integer from 0 through 3, with 0 indicating the highest privilege level.

Use the *entry-list* to specify individual table entries or groups of entries, with or without explicit indexes in the tables. The *entry-list* specification supplements and modifies the default table organizations. If an entry is installed in an LDT and in a GDT or IDT, references to that entry are resolved through the GDT or IDT.

Use the *res-list* to reserve table slots. BLD386 assigns selectors that correspond to these slots. It fills reserved slots with zeros. The *index2* must be greater than or equal to *index1*.

The *presence-indicator* must be 1, PRESENT, for any table that has a defined *base-addr*.

The *output-indicator* determines whether the table is written to the output object file. Use the NOT CREATED option to omit tables from the output object file. This option is useful for real mode applications that have the USEREAL attribute, and for flat model systems. NOT CREATED is valid only for a bootloadable object module, but CREATED is valid for both a bootloadable or a dynamically loadable object module (see USEREAL in the SEGMENT definition in this chapter and the FLAT and BOOTLOAD controls in Chapter 4).

Tables 3-7, 3-8, and 3-9 summarize the parameters used to define the *entry-list* for the GDT, the IDT, and any LDT.

TABLE Definition (continued)

Table 3-7 Entry-List Parameters for the GDT

Parameter	Definition	Values
<i>index</i>	The number of a slot in the GDT. By default, BLD386 assigns values for slots 0-2 for bootloadable modules.	1-8190
<i>entry</i>	<p>Any descriptor can be installed, except for Interrupt and trap gates. Segment descriptors must be for input segments, and gates must be defined already. TSS and LDT descriptors must refer to already-defined TSSs and LDTs.</p> <p>In addition to installing user-specified entries, BLD386 installs any of the following descriptors that are defined but not explicitly installed: <i>task-id</i>, <i>task-id.LDT</i>, and <i>anldt-id</i>.</p>	<i>seg-name</i> <i>gate-id</i> <i>task-id</i> <i>task-id.LDT</i> <i>anldt-id</i>

Table 3-8 Entry-List Parameters for the IDT

Parameter	Definition	Values
<i>index</i>	The number of a slot in the IDT. Slots 0-31 correspond to Intel-defined or Intel-reserved interrupts.	0-255
<i>entry</i>	Only trap, Interrupt or task gates can be installed. These must be defined earlier in the build program.	<i>gate-id</i>

TABLE Definition (continued)

Table 3-9 Entry-List Parameters for an LDT

Parameter	Definition	Values
<i>index</i>	The number that refers to a slot in an LDT. By default, BLD386 assigns values for slots 0-1.	0-8190
<i>entry</i>	<p>Any segment descriptor, call gate, or task gate can be installed in an LDT, if the segment is in an input module and the gate is defined earlier in the build program.</p> <p>In addition to installing user-specified entries, BLD386 installs segment descriptors, call gates, or task gates that are defined but not explicitly installed as follows: in the first user-defined LDT, or in a BLD386-defined LDT named LDT? if no LDT has been defined.</p>	<i>seg-name</i> <i>gate-id</i>

Examples

1. This example defines an LDT `task1ldt`. This table contains all the segments in the module `sample`. Its entries 0 through 10 are reserved, and it is at privilege level 0 by default.

```
TABLE                                --an LDT containing descriptors for
task1ldt (RESERVE = (0..10),         -- all segments from module sample is
ENTRY = (sample));                  -- created
```

2. This example defines a GDT and an IDT. It places the GDT at level 0. It contains all the segments in the `osmod` module, the task gate `taskentry`, and the task state segment descriptors `task1` and `task2`, along with the descriptor for the LDT. The GDT has 80 additional entries at the end of the table. Entries from 0 through 30 and from 50 through 55 are reserved. The GDT limit and base are stored at the location indicated by the public symbol `gdtloc`. The IDT is also at level 0 and contains the interrupt gate `errhndler` installed at slot number 50. Figure 3-9 represents the resulting GDT.

TABLE Definition (continued)

TABLE		--the GDT contains descriptors for
GDT	(BASE = 100H,	-- segments from osmod, task1's TSS,
	LIMIT = +80,	-- task2's TSS, the LDT common_ldt, and a task
	DPL = 0,	-- gate named taskentry
	RESERVE = (0..30, 50..55),	
	ENTRY = (osmod,	--the GDT has 80 extra entries and 37 slots
	common_ldt,	
	taskentry,	
	59:(task1, task2)),	
	LOCATION = gdtloc),	--the GDT limit and base are stored
		-- at gdtloc
IDT	(DPL = 0,	--the IDT contains an interrupt gate
	ENTRY = (50:errhndler));	-- called errhndler in slot number 50

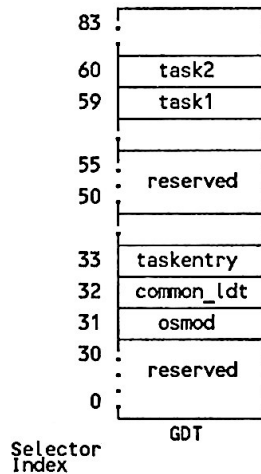


Figure 3-9 GDT Entries for Example 2

TASK Definition

Purpose and Scope

This definition creates task state segments (TSSs). Initialization information can be extracted from an input module or defined explicitly. For every TSS defined, BLD386 creates a task descriptor. You can override the default values for the task descriptor in the TASK definition.

The 80386 hardware supports TSSs in both 80286 and 80386 format. However, only 80386-format TSSs are generated by BLD386.

Syntax

task-definition ⇒ TASK *newtask* [...]

$$\text{newtask} \Rightarrow \text{task-id} \left(\left\{ \begin{array}{l} \text{BASE} = \text{base-addr} \\ \text{CODE} = \text{code-id} \\ \text{LDT} = \text{ldt-id} \\ \text{OBJECT} = \text{module1-id} \\ \text{DPL} = \text{priv-level} \\ \text{DATA} = \text{initial-data} \\ \text{STACKS} = (\text{stack-list}) \\ \text{LIMIT} = \text{task-seg-limit} \\ \text{task-attribute} \\ \text{debug-indicator} \\ \text{presence-indicator} \\ \text{task-flags} \end{array} \right\} \right) [\dots]$$

base-addr ⇒ { *number32* | AT (*public-id*) }

priv-level ⇒ { 0 | 1 | 2 | 3 }

$$\text{initial-data} \Rightarrow \left\{ \begin{array}{l} \text{module2-id} \\ [\text{module2-id.}] \text{combine1-id} \end{array} \right\}$$
$$\text{stack-list} \Rightarrow \left\{ \begin{array}{l} \text{module3-id} \\ [\text{module3-id.}] \text{combine2-id} \end{array} \right\} [\dots]$$

TASK Definition (continued)

task-seg-limit ⇒ [+] *number32*

task-attribute ⇒ [NOT] INITIAL

debug-indicator ⇒ [NOT] DEBUGTRAP

presence-indicator ⇒ [NOT] PRESENT

$$\text{task-flags} \Rightarrow \left\{ \begin{array}{l} \text{IOPRIVILEGE} - \{ 0 \mid 1 \mid 2 \mid 3 \} \\ \text{[NOT] INTENABLED} \\ \text{[NOT] VIRTUALMODE} \end{array} \right\}$$

Where:

- task-id* names the TSS. Use *task-id* to refer to task descriptors when installing them in a GDT and during creation of task gates (see the TABLE and GATE definitions in this chapter).
- When the input linkable modules contain a main module but the build program contains no *task-definition*, BLD386 creates a TSS named TASK? and extracts initial values from the main module. BLD386 also creates a public definition for each task in debug information.
- public-id* defines the *base-addr* of the TSS. The *public-id* must be a public identifier and must belong to a segment that has already been assigned its absolute address. The base of the TSS identified by *task-id* is set to the address of the *public-id*.
- code-id* specifies an entry point to the task, thus defining the initial CS and EIP (entry point) registers for the task. If the OBJECT specification is used, omit the CODE specification.
- ldt-id* names the related LDT for the task and places the selector for the LDT in the TSS. Note that descriptors for the LDTs themselves are always placed in the GDT.

TASK Definition (continued)

<i>module1-id</i>	names a module from which BLD386 can extract values otherwise specified using <i>code-id</i> , <i>initial-data</i> , and a subset of <i>stack-list</i> . The <i>module1-id</i> identifies an input main module. Omit the CODE specification if the OBJECT specification is used.
<i>initial-data</i>	points to a segment that indicates the initial DS register value. BLD386 determines the register values based on information in the input linkable module or the segment indicated by <i>initial-data</i> , or both. The <i>combine1-id</i> should be specified with a <i>module2-id</i> when more than one input segment may have the same name. If the OBJECT specification is used, omit the DATA specification.
<i>stack-list</i>	is one or more segment identifiers that point to a stack segment. These identifiers set the initial SS:ESP register values for each of the privilege levels at which the segments indicated by <i>module3-id</i> , <i>module3-id.combine2-id</i> , or <i>combine2-id</i> exist. As many as three segments can be specified in <i>stack-list</i> if they are at different privilege levels. The stack privilege level corresponds to the level of the segment specified. The initial corresponding SS:ESP (stack pointer) register value is stored in the TSS (see Figure 3-10).

Discussion

Figure 3-10 shows the format of a TSS.

64H	I/O Permission Base	Reserved	T
60H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LDT	
5CH	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	
58H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	
54H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	
50H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	
4CH	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	
48H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	
44H	EDI		
40H	ESI		
3CH	EBP		
38H	ESP		
34H	EBX		
30H	EDX		
2CH	ECX		
28H	EAX		
24H	EFLAGS		
20H	EIP		
1CH	CR3		
18H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	
14H	ESP2		
10H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	
0CH	ESP1		
8H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	
4H	ESP0		
0H	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Back Link	
	31	15	0

Figure 3-10 80386 TSS Format

TASK Definition (continued)

Use the **BASE** construct to override the default 32-bit base address for the TSS. When the **BOOTLOAD** control is in effect (see Chapter 4), BLD386 assigns an absolute value to the base if a base value specification is omitted. Specify the base address numerically or symbolically.

The *task-attribute* specifies whether this task is the initial task. If **INITIAL/NOT INITIAL** is omitted, the first task encountered in the build program is set to be the initial task by default.

DPL sets the TSS's privilege level in the task descriptor. The privilege level is checked when the current task uses a **JMP** or a **CALL** instruction to perform a task switch. The access rights of a **JMP** or **CALL** also apply when a **JMP** or **CALL** causes a task switch. However, these rights apply to the target TSS rather than the DPL of the target code segment. Note that the DPL of the TSS itself is different than the DPL of the *code-id* in the task. The *priv-level* can be any value from 0 through 3, with 0 the highest privilege level.

The *debug-indicator* specifies whether a debug exception is raised on a task switch. The T-bit in the high-order dword in the TSS is the debug trap flag. The default is 0, **NOT DEBUGTRAP**.

The **OBJECT** and/or **STACKS** constructs can define up to three additional **SS:ESP** values. One of the four stack segments should correspond to the DPL of *code-id*. BLD386 chooses the stack segment having the same privilege as that of *code-id* to determine the starting **SS:ESP** values for the task. The others are used as initial stacks to be used at the remaining privilege levels. The DPLs of the segments are specified in the **SEGMENT** definition.

BLD386 sets **SS** to point to the base of the segment. **ESP** is set as follows:

- For expanddown stack segments, **ESP** is 0.
- For non-expanddown stack segments, **ESP** is the segment size (limit + 1).
- For DSC (data/stack combined) segments, **ESP** points to the first byte above the stack portion of the combined segment. Figure 3-11 shows the organization of a DSC. See the **SEGMENT** definition in this chapter for more about DSC segments.

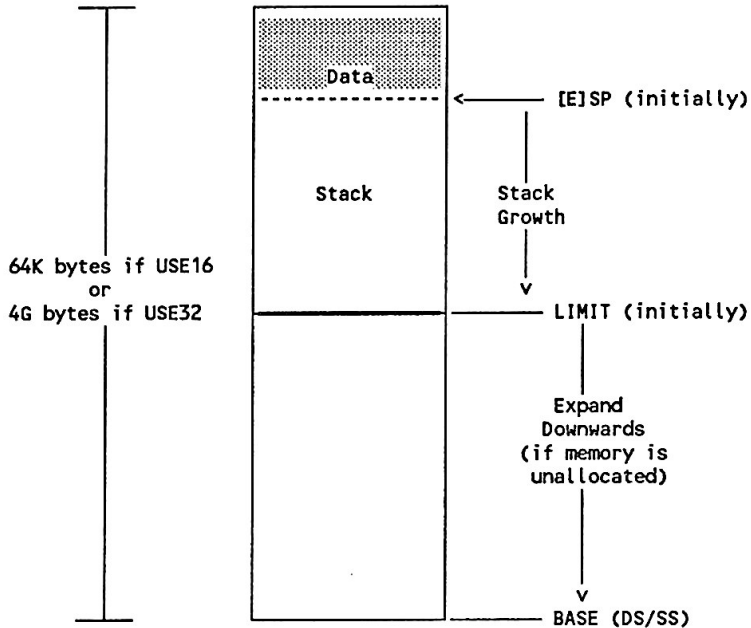


Figure 3-11 Data/Stack Combined Segment Organization

The *task-seg-limit* specifies the maximum size of the TSS. The default limit is 103 bytes. The maximum limit is 4G bytes-1. BLD386 creates only 80386-type TSSs.

The *presence-indicator* specifies whether the task is present or not present. If the *presence-indicator* PRESENT, the present bit in the task descriptor is 1; if NOT PRESENT, it is 0. By default, the bit is 1.

The *task-flags* construct specifies the initial values of three fields of the EFLAGS register: the I/O privilege level (IOPL), the interrupt status of the current task, and the setting of the VM bit. Figure 3-12 shows details of the EFLAGS register.

TASK Definition (continued)

The **IOPRIVILEGE** construct sets the IOPL value in the EFLAGS double word of the TSS. The IOPL value in the EFLAGS double word controls the current task's right to execute the privileged instructions, e.g., IN, INS, OUT, OUTS, CLI, STI, and LOCK. A task may not execute any privileged instructions if the DPL of its code segment is numerically greater than the IOPL. By default, the IOPL is 0.

The **INTENABLED** construct sets or clears the IF bit in the FLAGS word of the TSS. By default, the IF bit is true (IF = 1), to enable interrupts.

VIRTUALMODE controls setting of the VM bit in the EFLAGS register in the TSS. **VIRTUALMODE** is valid only with the **BOOTLOAD** control (see Chapter 4 for information). The 80376 does not support **VIRTUALMODE**.

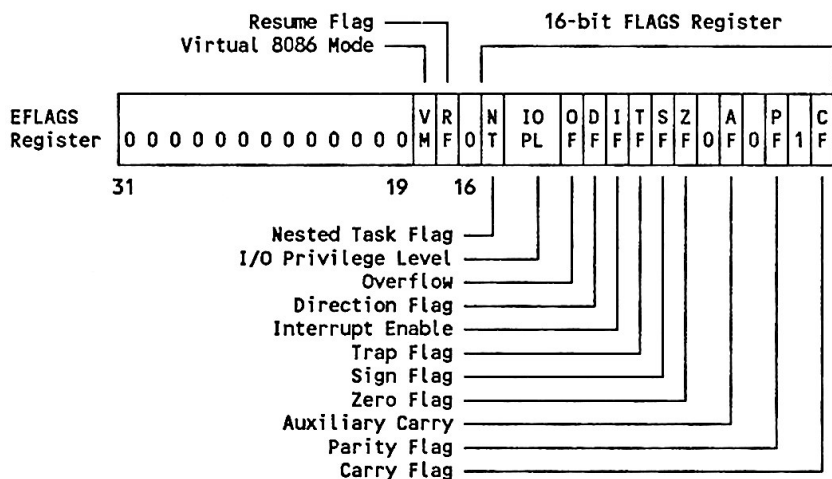


Figure 3-12 EFLAGS Register

TASK Definition (continued)

Table 3-10 summarizes the syntax for creating a task segment.

Table 3-10 Syntax Summary for Task State Segment

TSS Field	Definition Parameter	Values	Default
CS Selector, EIP (entry point)	<i>code-id</i>	CODE= <i>code-id</i>	-
Task LDT Selector	<i>ldt-id</i>	LDT= <i>ldt-id</i>	-
	<i>module1-id</i>	OBJECT= <i>module1-id</i>	-
DS Selector	<i>initial-data</i>	DATA={ <i>module2-id</i> [<i>module2-id</i> .]combine1-id}	-
SS Selectors	<i>stack-list</i>	STACKS={ <i>module3-id</i> [<i>module3-id</i> .]combine2-id}	-
T	<i>debug-indicator</i>	[NOT] DEBUGTRAP	NO DB
EFLAGS: I/O privilege Interrupt Virtual mode	<i>task-flags</i>	IOPRIVILEGE={0 1 2 3} [NOT] INTENABLED [NOT] VIRTUALMODE	IP=0 IE NO VM

TASK Definition (continued)

Table 3-11 summarizes the syntax for creating a task descriptor.

Table 3-11 Syntax Summary for Task Descriptor

Descriptor Field	Definition Parameter	Values	Default
Base	<i>base-addr</i>	BASE={ <i>number32</i> AT (<i>public-id</i>)}	*
DPL	<i>priv-level</i>	DPL={0 1 2 3}	DPL=0
Limit	<i>task-seg-limit</i>	LIMIT=[+] <i>number32</i>	103 (67H)
P	<i>presence-indicator</i>	[NOT] PRESENT	PRESENT

Notes:

*BLD386 assigns a value to this field if the BOOTLOAD control is in effect (see Chapter 4).

NOTE

If there is no main module in the input files and no *task-definition* in the build program, BLD386 issues a warning and creates no TSS.

Examples

1. This task definition shows an example where all the information for the creation of the task state segment is extracted from the object module `sample_mod`.

TASK

```
sample_task (OBJECT = sample_mod);    --all TSS information is extracted
                                      -- from sample_mod
```

TASK Definition (continued)

2. This example shows a definition of an LDT table `commonldt`, and two tasks `task1` and `task2`. Each of the tasks executes code from a different module, but they share the same LDT.

TABLE

```
COMMONLDT  (ENTRY = (TASK1MOD,      --the LDT and its entries
                TASK2MOD,
                DQATTACH,
                SYSGATE));
```

TASK

```
task1      (OBJECT = task1_mod,      --two tasks are defined, and they
            LDT    = common_ldt),    -- share the same LDT, common_ldt, which
task2      (OBJECT = task2_mod,      -- was defined in an earlier
            LDT    = common_ldt);    -- table definition
```

3. This example shows the specification of the `IOPRIVILEGE` and the interrupt enable flags that are placed in the task state segments for `task11` and `task21`.

TASK

```
task11     (IOPRIVILEGE = 2,        --sets IOPL at 2 and IF to false
            NOT INTENABLED),
task21     (IOPRIVILEGE = 0,        --sets IOPL at 0 and IF to true
            INTENABLED);
```

3.3 Build Program Examples

This example shows an entire build program. It includes definitions for a system that consists of a kernel, module `sysmodule`, and run-time utilities. modules `iomodule` and `utilmodule`. Four gates are defined: three call gates and one interrupt gate. For the single TSS defined, BLD386 extracts initialization information from module `iomodule`. The stacks defined for the TSS have the same *combine-id*, so their respective *module-ids* are given.

`sysgenbuildfile;`

SEGMENT

```
sysmodule (DPL = 0),  
iomodule (DPL = 1),  
utilmodule (DPL = 1);
```

GATE

```
createfile,  
openfile,  
closefile,  
errorhandler (INTERRUPT);
```

TASK

```
systask (OBJECT = iomodule,  
STACKS = (sysmodule.stack,  
iomodule.stack));
```

TABLE

```
GDT (ENTRY = (systask,  
sysmodule,  
iomodule,  
utilmodule,  
createfile,  
openfile,  
closefile)),  
IDT (ENTRY = 2:errorhandler);
```

END

The following build program defines two tasks that share the same LDT and a global data area. The LDT, named `commonldt`, contains descriptors for all segments in modules `mod1` and `mod2`. The global data area is defined in module `sdata`. The two task state segments,

called `task1block` and `task2block`, each have a stack segment defined in module `nucleus`. The GDT houses the LDT descriptor, two TSS descriptors, two task gates called `task1gate` and `task2gate`, and descriptors for all segments in the module `nucleus`.

```
multitask;
```

```
    SEGMENT
```

```
        nucleus (DPL = 0);
```

```
    TABLE
```

```
        commonldt (ENTRY = (mod1, mod2, sdata));
```

```
    TASK
```

```
        task1block (OBJECT = mod1,          --DPL=3
                    LDT = commonldt,
                    STACKS = (nucleus.stack1)), --DPL=0
```

```
        task2block (OBJECT = mod2,          --DPL=3
                    LDT = commonldt,
                    STACKS = (nucleus.stack2)); --DPL=0
```

```
    GATE
```

```
        task1gate (TASK,
                   ENTRY = task1block);
        task2gate (TASK,
                   ENTRY = task2block);
```

```
    TABLE
```

```
        GDT (ENTRY = (nucleus,
                      commonldt,
                      task1block,
                      task1gate,
                      task2block,
                      task2gate));
```

```
END
```



Contents

Chapter 4 Invocation

4.1	DOS Invocation Syntax	4-1
4.2	VMS Invocation Syntax	4-2
4.3	Console Messages	4-4
4.4	Control Files	4-5
4.5	BLD386 Controls.....	4-6
	BOOTLOAD	4-13
	BOOTSTRAP.....	4-15
	BUILDFILE.....	4-17
	CONTROLFILE	4-19
	DEBUG.....	4-21
	ERRORPRINT	4-23
	FILL.....	4-25
	FLAT	4-27
	LIST	4-31
	MAP.....	4-32
	MOD376 or MOD386.....	4-33
	OBJECT	4-35
	PAGETABLES	4-37
	PRINT.....	4-38
	RELDISC.....	4-40
	TITLE	4-41
	TYPE.....	4-43
	WARNINGS.....	4-44



This chapter describes how to use BLD386 and contains the following main topics:

- Invocation
- Console messages
- Control files
- Controls

4.1 DOS Invocation Syntax

To invoke BLD386 on a DOS operating system, follow this syntax:

BLD386 *input-list* [*control* ...]

$$\textit{input-list} \Rightarrow \left\{ \begin{array}{l} \textit{filename} \\ \textit{filename} (\{ \textit{module-list} \mid * \}) \end{array} \right\} [, \dots]$$

module-list \Rightarrow *module-name* [, ...]

Where:

input-list is one or more linkable modules or object library modules for BLD386 to process.

control is a specification defined in Section 4.5 of this chapter.

filename is a character string by which the operating system can identify a file, including the pathname where necessary. If an asterisk (*) is specified in place of *module-list*, BLD386 processes all modules of the file named by *filename*.

module-name is a character string identifying a module.

If *module-names* are specified with the library *filename*, BLD386 processes only those modules. If an asterisk (*) is specified after a library *filename*, BLD386 processes all the modules in the library. If no *module-name* is specified after the *filename* of a library file, then the library file is processed only if there is at least one unresolved external symbol. The library is scanned for modules containing public symbols that match the as yet unresolved external symbols. Each such module is processed as if it were explicitly specified. The selection process continues until the modules in the library cannot satisfy any more unresolved external symbols (including any external symbols encountered while processing modules from the library).

If no *module-name* is specified after the *filename* of a linkable file, BLD386 processes all modules in the file named by *filename*, as if an asterisk (*) were specified.

Duplicate *module-names* constitute an error.

The *input-list* can be omitted from the invocation if it is specified in one or more control files (see Section 4.4 for information about the structure of control files).

Continue the invocation line by entering a continuation character, the ampersand (&), before entering the line terminator. The continuation line then appears with the DOS system prompt character.

4.2 VMS Invocation Syntax

To invoke BLD386 on a VMS operating system, follow this syntax:

BLD386 [*control ...*] *input-list*

$$\text{input-list} \Rightarrow \left\{ \begin{array}{l} \text{filename} \\ \text{"filename (\{ module-list | * \})"} \end{array} \right\} [, \dots]$$

module-list \Rightarrow *module-name* [, ...]

Where:

<i>control</i>	is a specification defined in Section 4.5 of this chapter.
<i>input-list</i>	is one or more linkable modules or object library modules for BLD386 to process.
<i>filename</i>	is a character string by which the operating system can identify a file, including the device name and directory where necessary. If an asterisk (*) is specified in place of <i>module-list</i> , BLD386 processes all modules in <i>filename</i> .
<i>module-name</i>	is a character string identifying a module.

If *module-names* are specified with the library *filename*, BLD386 processes only those modules. If an asterisk (*) is specified after a library *filename*, BLD386 processes all the modules in the library. If no *module-name* is specified after the *filename* of a library file, then the library file is processed only if there is still at least one unresolved external symbol. The library is scanned for modules containing public symbols that match the as yet unresolved external symbols. Each such module is processed as if it were explicitly specified. The selection process continues until the modules in the library cannot satisfy any more unresolved external symbols (including any external symbols encountered while processing modules from the library).

If no *module-name* is specified after the *filename* of a linkable file, BLD386 processes all modules in the file named by *filename*, as if an asterisk (*) were specified.

Duplicate *module-names* constitute an error.

The *input-list* can be omitted from the invocation if it is specified in one or more control files (see Section 4.4 for information about the structure of control files).

Continue the invocation line by entering a continuation character, the ampersand (&), before entering the line terminator. The continuation line then appears with the VMS system prompt character.

NOTE

BLD386 processes the files in the input list in the order they are specified. Therefore, BLD386 uses public symbols in a library file to resolve only those external symbols known to be unresolved at the time of the examination of the library file.

4.3 Console Messages

Two kinds of messages appear at the console during a BLD386 operation: sign-on/sign-off messages and fatal error messages.

BLD386 signs on to the console after invocation as follows:

```
system-id 80386 SYSTEM BUILDER, version  
Copyright years Intel Corporation
```

Where:

system-id is the string returned by the operating system.

version is the version number of BLD386.

years indicates the copyright year or years.

BLD386 signs off as follows when it completes its processing without fatal errors:

```
PROCESSING COMPLETED.      m WARNINGS,      n ERRORS
```

Where:

m and *n* represent the number of warning and nonfatal error conditions, respectively, that occurred during processing. The ERRORPRINT control directs BLD386 to display warning and error messages to the console for DOS, to SYSS\$OUTPUT for VMS, or to a specified file. These numbers can be affected by the NOWARNINGS control (see Section 4.5 in this chapter).

If BLD386 encounters a fatal error condition, it sends the following sign-off message to the console:

PROCESSING ABORTED

Fatal error messages are always displayed at the console. See Appendix C for error message explanations and suggestions for correcting the error condition.

4.4 Control Files

A control file is a text file that can contain any file names and controls that normally appear in the invocation line. For example, an invocation line that contains five controls can be simplified by placing the controls in a single control file and directing BLD386 to process the control file. Specify a control file in the invocation line, as follows:

DOS:

```
BLD386 CONTROLFILE (filename [...])
```

VMS:

```
{ BLD386/CONTROLFILE = filename  
  BLD386/CONTROLFILE = ( filename [...] )  
  BLD386 "CONTROLFILE ( filename [...] )" }
```

Where:

filename is a file that can contain input files and any controls except CONTROLFILE. Control files that contain only controls can be specified in any position in the control list. Those that contain only input files for the *input-list* can be specified anywhere in the *input-list*. Within those that contain both input files for the *input-list* and controls, all input file names must be listed before the controls.

VMS: When a control file is specified in the *input-list*, the entire specification must be enclosed in double quotes ("), and the list of *filenames* must be enclosed in parentheses.

Controls can be written in uppercase or lowercase. When reading a control file, BLD386 ignores characters between a semicolon or a line continuation character and a line terminator. Use a semicolon to introduce comments. A line continuation character indicates that the control file continues on the next line.

The line terminator is treated like a blank if it follows a line continuation character. Lines in a control file cannot exceed 120 characters in length. An example control file that contains only file names for the input list is as follows:

```
util.lib, & ; utility library
system lib ; system library
```

An example control file that contains the last file names for the input list and all of the controls is as follows:

```
util.lib, & ; utility library
system.lib & ; system library
nobl & ; loadable module
li & ; include buildfile listing in print file
ma & ; include maps in print file
oj (lbt.sys) ; name output file
```

4.5 BLD386 Controls

This section lists the BLD386 controls in alphabetical order. Each control is described in detail. Use BLD386 controls to select input and output files, the build program file, and control files. BLD386 controls determine the type of build performed, various aspects of the built system, and the information sent to print and map files.

Tables 4-1 (for DOS) and 4-2 (for VMS) summarize the controls as used on the command line. Most controls have an optional NO prefix. The default column shows the condition in effect if a control is not specified. Table 4-3 lists abbreviations for each control. If one invocation contains duplicate controls, BLD386 processes only the rightmost specification. For example, if NOBOOTLOAD is specified early in the invocation line, and BOOTLOAD is specified later, NOBOOTLOAD is overridden.

VMS NOTE

Any controls can also be abbreviated if the first four characters are unique. Otherwise, the name must be written in full or abbreviated according to the abbreviations given with the syntax. For example, **BOOTSTRAP** and **BOOTLOAD** cannot be abbreviated according to VMS notation, but can be abbreviated **BS** and **BL**, respectively. The **ERRORPRINT** control can be abbreviated as either **ER** or **EP**. VMS does not recognize the full name and the abbreviations as the same control, so do not to mix abbreviations and full names in the same invocation.

Table 4-1 Summary of BLD386 Controls for DOS

Command Line Syntax	Description	Default
BOOTLOAD NOBOOTLOAD	Assigns or does not assign absolute addresses	BOOTLOAD
BOOTSTRAP (<i>symbol-name</i> <i>number</i>)	Places a jump instruction at location 0FFFFFFF0H for system boot	
BUILDFILE [(<i>filename</i>)] NOBUILDFILE	Identifies buildfile	BUILDFILE (<i>first-input-filename</i> .BLD)
CONTROLFILE (<i>filename</i> [...])	Specifies file for input elements	
DEBUG NODEBUG	Retains or removes debug information	DEBUG
ERRORPRINT [(<i>filename</i>)] NOERRORPRINT	Names or suppresses naming of error print file	NOERRORPRINT
FILL [(<i>n</i>)] NOFILL	Fills with zeros or <i>n</i> , or suppresses filling uninitialized areas	FILL
FLAT	Creates overlapping 4G byte code and data segments	
LIST NOLIST	Includes or suppresses build file listing	LIST
MAP NOMAP	Includes or suppresses map listings	MAP

Table 4-1 Summary of BLD386 Controls for DOS (continued)

Command Line Syntax	Description	Default
MOD376 MOD386	Specifies target processor	MOD386
OBJECT [(<i>filename</i>)] NOOBJECT	Names or suppresses naming object module output	OBJECT (<i>first-input-filename</i>)
PAGETABLES NOPAGETABLES	Generates pagetables if BOOTLOAD and MOD386 are in effect	NOPAGETABLES
PRINT [(<i>filename</i>)] NOPRINT	Names or suppresses naming of print file	PRINT (<i>output-object-filename</i> .MP2)
RELDESC	Outputs relocation information for GDT entities not explicitly installed in slots	
TITLE (<i>string</i>)	Places header line at the top of each print file page	TITLE (<i>null-string</i>)
TYPE NOTYPE	Enables or suppresses type checking	TYPE
WARNINGS NOWARNINGS [(<i>n</i> [...])]	Includes or suppresses (by number, if specified) error and warning messages	WARNINGS

Table 4-2 Summary of BLD386 Controls for VMS

Command Line Syntax	Description	Default
/BOOTLOAD /NOBOOTLOAD	Assigns or does not assign absolute addresses	/BOOTLOAD
/BOOTSTRAP= {symbol-name number}	Places a jump instruction at location 0FFFFFF0H for system boot	
/BUILDFILE[=filename] /NOBUILDFILE	Identifies buildfile	/BUILDFILE=first- input-filename .BLD
/CONTROLFILE={filename (filename [...])}	Specifies file for input elements	
/DEBUG /NODEBUG	Retains or removes debug information	/DEBUG
/ERRORPRINT[=filename] /NOERRORPRINT	Names or suppresses naming of error print file	/NOERRORPRINT
/FILL[=n] /NOFILL	Fills with zeros or <i>n</i> , or suppresses filling uninitialized areas	/FILL
/FLAT	Creates overlapping 4G byte code and data segments	
/LIST /NOLIST	Includes or suppresses build file listing	/LIST
/MAP /NOMAP	Includes or suppresses map listings	/MAP

Table 4-2 Summary of BLD386 Controls for VMS (continued)

Command Line Syntax	Description	Default
/MOD376 /MOD386	Specifies target processor	/MOD386
/OBJECT[= <i>filename</i>] /NOOBJECT	Names or suppresses naming object module output	/OBJECT= <i>first-input-filename</i> .DAT
/PAGETABLES /NOPAGETABLES	Generates pagetables if /BOOTLOAD and /MOD386 are in effect	/NOPAGETABLES
/PRINT[= <i>filename</i>] /NOPRINT	Names or suppresses naming of print file	/PRINT= <i>output-object-filename</i> .MP2
/RELDESC	Outputs relocation information for GDT entities not explicitly installed in slots	
/TITLE= <i>string</i>	Places header line at the top of each print file page	/TITLE= <i>null-string</i>
/TYPE /NOTYPE	Enables or suppresses type checking	/TYPE
/WARNINGS /NOWARNINGS[=(<i>n</i> [,...])]	Includes or suppresses (by number, if specified) error and warning messages	/WARNINGS

Table 4-3 Standard Abbreviations for BLD386 Controls

Word	Abbr.	Word	Abbr.
BOOTLOAD	BL	NODEBUG	NODB
BOOTSTRAP	BS	NOERRORPRINT	NOEP
BUILDFILE	BF	NOFILL	NOFI
CONTROLFILE	CF	NOLIST	NOLI
DEBUG	DB	NOOBJECT	NOOJ
ERRORPRINT	EP	NOPRINT	NOPR
FILL	FI	NOTYPE	NOTY
FLAT	FL	NOWARNINGS	NOWA
LIST	LI	OBJECT	OJ
MAP	MA	PAGETABLES	PT
MOD376	M376	PRINT	PR
MOD386	M386	RELDISC	RD
NOBOOTLOAD	NOBL	TITLE	TT
NOBOOTSTRAP	NOBS	TYPE	TY
		WARNINGS	WA

BOOTLOAD

Designates bootloadable
or loadable output

Syntax

DOS: BOOTLOAD
NOBOOTLOAD

VMS: /BOOTLOAD
/NOBOOTLOAD

Abbreviation

BL
NOBL

Default

BOOTLOAD

Description

BOOTLOAD directs BLD386 to produce a bootloadable object module. BLD386 assigns default absolute addresses to all tables, segments, and TSSs. The default absolute addresses can be overridden in the build program. For tables and TSSs, the default alignment is paragraph. For segments, the language translator sets the default alignment; the build program can override this alignment.

NOBOOTLOAD suppresses automatic addressing and directs BLD386 to produce a loadable object module.

NOTES

BLD386 checks to ensure enough memory space is available when it assigns absolute addresses.

Bootloadable modules always contain a GDT and an IDT. The first 32 interrupts are Intel-reserved. By default, BLD386 creates an IDT with 32 descriptors. Use the build file TABLE definition (see Chapter 3) to override this default.

Examples

1. In this example, BLD386 produces a bootloadable object module as output from the named linkable input modules MODA and MODB, which reside in the input library file MOD.LIB. The output file name is MOD.

DOS: BLD386 MOD.LIB (MODA,MODB) BOOTLOAD

VMS: BLD386/BOOTLOAD "MOD.LIB(MODA,MODB)"

2. In this example, BLD386 produces a loadable object module as output from all linkable input modules in the input file MOD.OBJ. The output file name is MOD.

DOS: BLD386 MOD.OBJ OBJECT (MOD.DAT) NOBOOTLOAD

VMS: BLD386/NOBOOTLOAD/OBJECT=MOD.DAT MOD.OBJ

BOOTSTRAP

Places near jump at
address 0FFFFFFF0H for
system boot

Syntax

DOS: `BOOTSTRAP (symbol-name | number)`

VMS: `/BOOTSTRAP = { symbol-name | number }`

Abbreviation

BS

Default

No near jump instruction is placed at 0FFFFFFF0H.

Description

BOOTSTRAP causes BLD386 to place a near jump instruction at location 0FFFFFFF0H to the location designated by *symbol-name* or *number*. A *symbol-name* must be a public symbol in the input. A *number* must represent a 32-bit absolute address. In either case, BLD386 places the jump instruction at location 0FFFFFFF0H, pointing to the designated address.

The jump instruction uses a 16-bit offset if the MOD386 control is also specified, or a 32-bit offset if the MOD376 control is also specified.

Use BOOTSTRAP to help write the necessary instructions that bring up the 80386 processor from real address mode to protected mode. This control is effective only if the BOOTLOAD control is in effect.

NOTES

At 80386 reset, the initial values of the CS and IP registers are 0F000H and 0FFF0H, respectively. Combining these values gives an initial starting address of 0FFFF0H. However, the address lines A20 through A31 of the 80386 are set high giving an effective starting address of 0FFFFFF0H (even though such an address is out of the 80386 range in real mode). A20 through A31 remain high until the first far call or jump is executed, when they drop to low.

BLD386 checks that the address specified as *symbol-name* or *number* is within the top 64K bytes of memory (0FFFF000H to 0FFFFFFCH) and issues an error message if it is not.

Examples

1. In this example, BLD386 produces a bootloadable object module as output from the object modules in the input file MOD.OBJ. A near jump instruction with the symbolic address START3 as its target is generated and placed at location 0FFFFFF0H.

DOS: BLD386 MOD.OBJ BOOTSTRAP (START3) BOOTLOAD

VMS: BLD386/BOOTLOAD/BOOTSTRAP=START3 MOD.OBJ

2. In this example, BLD386 produces a bootloadable object module as output from the object modules in the input file MOD.OBJ. A near jump instruction is placed at location 0FFFFFF0H with the address 0FFFFFF00H as its target.

DOS: BLD386 MOD.OBJ BOOTSTRAP (0FFFFFF00H) BOOTLOAD

VMS: BLD386/BOOTLOAD/BOOTSTRAP=0FFFFFF00H MOD.OBJ

BUILDFILE

Designates file containing
build language definitions

Syntax

DOS: BUILDFILE [(*filename*)]
NOBUILDFILE

VMS: /BUILDFILE [= *filename*]
/NOBUILDFILE

Abbreviation

BF
NOBF

Default

BUILDFILE using the first input file whose name has a .BLD extension, or NOBUILDFILE if there is no input file that has a .BLD extension.

Description

BUILDFILE identifies the file containing a build program (see Chapter 3). If BUILDFILE is specified without a file name, or if neither BUILDFILE nor NOBUILDFILE is specified, BLD386 assumes that the build program is contained in the file that has a default file name. The default file name is the first input file whose name has an extension of .BLD. If such a default file does not exist, NOBUILDFILE is assumed.

NOBUILDFILE specifies that no build program file is to be used in processing. BLD386 uses defaults to create the object file. Chapter 4 explains these defaults.

BUILDFILE (continued)

NOTE

If the specified *filename* matches that of an output file, a file in the input list, or a control file, BLD386 aborts processing.

Examples

1. In this example, BLD386 processes the build program from build file MOD.BLD.

DOS: BLD386 MOD.OBJ BUILDFILE (MOD.BLD)

VMS: BLD/BUILDFILE=MOD.BLD MOD.OBJ

2. In this example, BLD386 does not attempt to process any build file.

DOS: BLD386 MOD.OBJ NOBUILDFILE

VMS: BLD386/NOBUILDFILE MOD.OBJ

Syntax

DOS: CONTROLFILE (*filename* [,...])

VMS single file: /CONTROLFILE = *filename*

VMS one or more files: /CONTROLFILE = (*filename* [,...])

VMS input list element: "CONTROLFILE (*filename* [,...])"

Abbreviation

CF

Default

No control file is used.

Description

CONTROLFILE directs BLD386 to the specified file for controls or elements of the input list. BLD386 resumes processing the command line when it encounters the end of a control file.

See Section 4.4 for the content and format of control files.

NOTES

Do not use nested control files.

Do not use a partial control or partial input list element in a control file.

CONTROLFILE (continued)

Example

In this example, assuming that control file CNTL1.DAT contains UTIL.LIB and SYSTEM.LIB, and that control file CNTL2.DAT contains PRINT(SAMPLE.MAP) and DEBUG, the following invocation lines are equivalent:

DOS: BLD386 SAMPLE.OBJ, CF (CNTL2.DAT) CF (CNTL1.DAT)

DOS: BLD386 SAMPLE.OBJ, UTIL.LIB, SYSTEM.LIB PRINT (SAMPLE.MAP) DEBUG

VMS: BLD386/CONTROLFILE=(CNTL2.DAT) SAMPLE.OBJ,"CONTROLFILE(CNTL1.DAT)"

VMS: BLD386/PRINT=SAMPLE.MAP/DEBUG SAMPLE.OBJ,UTIL.LIB,SYSTEM.LIB

DEBUG

Retains or removes
debug information

Syntax

DOS: DEBUG
NODEBUG

VMS: /DEBUG
/NODEBUG

Abbreviation

DB
NODB

Default

DEBUG

Description

DEBUG places information used by symbolic debuggers and MAP386 into the loadable or bootloadable output module. Debug information can include symbolic names and line numbers generated by translators called with DEBUG, as well as public symbol information formatted for debuggers.

NODEBUG prevents symbolic debugging information from being placed in the output module.

The MAP386 utility requires debug information. See the *Intel386™ Family Utilities User's Guide* for information about MAP386.

DEBUG (continued)

Example

In this example, BLD386 places information to be used by symbolic debuggers into the bootloadable output object module MOD.DAT.

DOS: BLD386 MOD.OBJ OBJECT (MOD.DAT) DEBUG

VMS: BLD386/DEBUG/OBJECT=MOD.DAT MOD.OBJ

Syntax

DOS: `ERRORPRINT [(filename)]`
`NOERRORPRINT`

VMS: `/ERRORPRINT [= filename]`
`/NOERRORPRINT`

Abbreviation

EP
NOEP

Default

NOERRORPRINT

Description

ERRORPRINT directs error and warning messages to the console, or to the file *filename*.

NOERRORPRINT prevents error messages from being placed in a separate file.

Error and warning messages can be selectively suppressed from *filename* (see the [NO]WARNINGS control later in this chapter).

ERRORPRINT (continued)

NOTES

The specified *filename* must not be the same as another output file, or a file in the input list, or a control file. If it is, BLD386 processing aborts.

Regardless of the setting of these controls, fatal error messages are displayed at the console for DOS or sent to SYSS\$OUTPUT for VMS, all error and warning messages are included in the print file (see Chapter 5), and all error and warning messages are counted in the sign-off message.

Example

In this example, BLD386 creates a separate file MOD.LIS that contains all warning and error messages.

DOS: BLD386 MOD.OBJ ERRORPRINT (MOD.LIS)

VMS: BLD386/ERRORPRINT=MOD.LIS MOD.OBJ

FILL

Fills or suppresses
filling uninitialized areas

Syntax

DOS: FILL [(*n*)]
NOFILL

VMS: /FILL [= *n*]
/NOFILL

Abbreviation

FI
NOFI

Default

FILL

Description

FILL directs BLD386 to fill all uninitialized areas of segments with zeros or with the specified hexadecimal value *n*, if specified.

NOFILL indicates that uninitialized areas of memory or input segments should remain uninitialized in the output object module. However, Block Symbol Storage (BSS) areas of segments are always initialized to zeros for bootloadable output. For loadable output, up to 8K bytes of BSS are initialized to zeros, and the loader is responsible for the initialization of any additional bytes.

NOTE

FILL creates a full memory image. Insufficient disk space causes a file input/output error before the output module is complete.

FILL (continued)

Examples

1. In this example, BLD386 produces a bootloadable object module MOD.X from the two linkable files MOD.LNK and CPROG.OBJ. Uninitialized areas of segments are not filled. (BSS, if any, is still initialized to zeros.)

DOS: BLD386 MOD.LNK, CPROG.OBJ OBJECT (MOD.X) BOOTLOAD NOFILL

VMS: BLD386/OBJECT=MOD.X/BOOTLOAD/NOFILL MOD.LNK,CPROG.OBJ

2. In this example, the bootloadable object module MOD.X this time has uninitialized areas of segments filled with the binary byte pattern 10101010.

DOS: BLD386 MOD.LNK, CPROG.OBJ OBJECT (MOD.X) BOOTLOAD FILL (0AAH)

VMS: BLD386/OBJECT=MOD.X/BOOTLOAD/FILL=0AAH MOD.LNK,CPROG.OBJ



Syntax

DOS: FLAT

VMS: /FLAT

Abbreviation

FL


Default

If the FLAT control is not specified then memory is not automatically configured for flat model.



Description

Flat model is the simplest addressing scheme. FLAT directs BLD386 to configure a bootloadable file in flat model; i.e., there is one code segment and one data segment with default addressing from 0 to 4 gigabytes-1 (or from 0 to 16 megabytes-1 for the 80376). These are overlaid segments, and their descriptors allow access with no protection to all memory as one physical address space. The default DPL for these segments is 0. Base addresses and limits for these segments can be overridden with the SEGMENT definition (see Chapter 3), but both base addresses must be equal. BLD386 default memory allocation begins at the common base address of these segments. These segments can have memory addresses which wrap from high memory to low memory.



The code segment has the name `__phantom_code__` and the data segment has the name `__phantom_data__`. The descriptors for these segments are put in the GDT (see the TABLE definition in Chapter 3). See Chapter 6 for how to build a simple system configured in flat model.

FLAT (continued)

FLAT tells BLD386 to combine original input segments into `__phantom_code__` and `__phantom_data__` segments. Descriptors for original segments are invalid because original segments are combined into the phantom segments. This means that symbols are not based on original descriptors. All symbols in original segments are based on phantom descriptors.

BLD386 assigns default addresses. To override this assignment, specify base addresses for input segments, tables, or tasks in the build program (see Chapter 3). BLD386 adjusts offsets to be relative to the phantom segments and replaces all references to the descriptors of the original entities with references to the descriptors of the phantom segments. BLD386 also adjusts debug information relative to the phantom descriptors.

Simple flat model uses the defaults set up by BLD386: phantom segments are at privilege level 0, based at 0, have a limit of 4G bytes - 1, and have their descriptors in the GDT. The default DPL can be overridden by specifying the phantom segment in a build file SEGMENT definition. It is unnecessary to define the GDT with a TABLE definition in the build program (the descriptors of the original segments need not be placed in the GDT because they are not used).

In flat model, by default, all stack segments are not expanddown and are based relative to `__phantom_data__`. The simple flat model provides no protection. To protect data from a stack overflow, specify an expanddown stack as follows:

- Use the TABLE definition in the build program to install the descriptor for the expanddown stack segment in the GDT or in an LDT (see Chapter 3).
- Use the SEGMENT definition in the build program to specify a limit less than 4G bytes - 1 for `__phantom_code__` and `__phantom_data__` (see Chapter 3).
- Do not assign a base address for the stack segment in the build program. BLD386 assigns the base address to be the same as the base address of the phantom segments, as required for flat model.

- Use the SEGMENT definition in the build program to specify a limit less than 4G bytes - 1 for the expanddown stack segment (see Chapter 3).
- Do not define the expanddown stack segment as a data/stack combined (DSC) segment. The purpose of the expanddown stack is to protect the data from stack overflow. See the SEGMENT definition in Chapter 3 for more on data/stack combined segments.

If any of the preceding requirements is not met, BLD386 merges the stack into the `_phantom_data_` segment. Figure 4-1 represents the simple flat model.

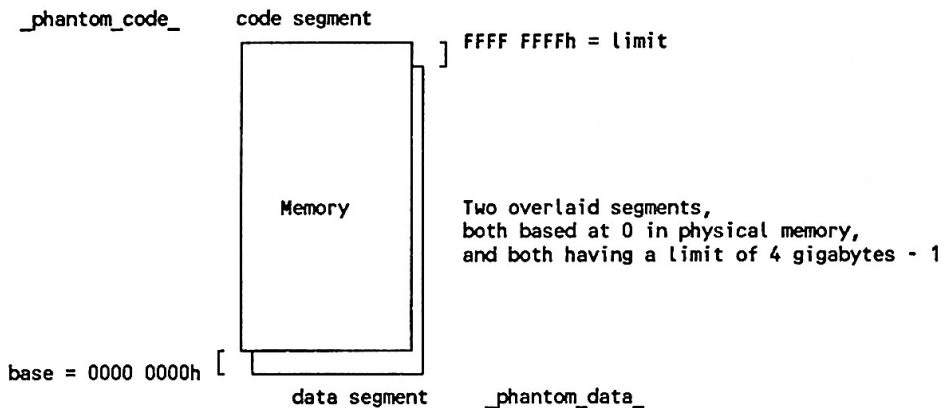


Figure 4-1 80386 Flat Memory Model

NOTES

BLD386 does not overlap any segments except `__phantom_code__` and `__phantom_data__` (unless the build program directs it to do so), but follows the memory allocation scheme described in Chapter 2.

Because original input segments are combined into `__phantom_code__` and `__phantom_data__`, BLD386 automatically changes the DPL for the original segments to match the DPL of the phantom segments.

Example

In this example, BLD386 is directed to create a bootloadable object module called SIMPLE configured in flat mode.

DOS: BLD386 BOOTLOAD FLAT SIMPLE.OBJ

VMS: BLD386/BOOTLOAD/FLAT SIMPLE.OBJ

LIST

Includes or suppresses
listing of build file

Syntax

DOS: LIST
NOLIST

VMS: /LIST
/NOLIST

Abbreviation

LI
NOLI

Default

LIST

Description

LIST directs BLD386 to include a build program listing in the print file.

NOLIST suppresses the appearance of a build program listing in the print file. Only build program records associated with an error condition are included in the print file.

The format of the build program listing is discussed in Chapter 5.

Example

In this example, BLD386 includes a build program listing for MOD.BLD in the print file MOD.LIS.

DOS: BLD386 MOD.OBJ PRINT (MOD.LIS) LIST

VMS: BLD386/LIST/PRINT=MOD.LIS MOD.OBJ

Invocation

MAP

Includes or suppresses
map listings

Syntax

DOS: MAP
NOMAP

VMS: /MAP
/NOMAP

Abbreviation

MA
NOMA

Default

MAP

Description

MAP directs BLD386 to include segment, gate, page table, and TSS maps of the output object module in the print file.

NOMAP suppresses inclusion of the maps in the print file.

Chapter 5 describes the print file map format.

Example

In this example, BLD386 includes maps for segments, gates, and TSSs in the print file MOD.LIS.

DOS: BLD386 MOD.OBJ PRINT (MOD.LIS) MAP

VMS: BLD386/MAP/PRINT=MOD.LIS MOD.OBJ

Syntax

DOS: MOD376
MOD386

VMS: /MOD376
/MOD386

Abbreviation

M376
M386

Default

MOD386

Description

MOD386 directs BLD386 to issue messages to guide creation of a file that can execute on the 80386. The 80386 processor allows the programmer to use all of the 80386 32-bit protection features, 80286-compatible segments (USE16), and 8086-compatible segments (USEREAL or VIRTUALMODE).

MOD376 directs BLD386 to issue messages to guide creation of a file that can execute on the 80376. The 80376 supports the 32-bit segmented model in protected mode only. Paging is not available. The 80376 has a 24-bit address bus; therefore, addresses which exceed the maximum physical address of 16M bytes - 1 wrap over to low memory. The 80376 does not support 80286 call, interrupt, or trap gates, or 80286 TSSs.

If the BOOTSTRAP control is specified with MOD376, the near jump instruction placed at the reset vector has a 32-bit offset. If MOD386 is specified, the offset is 16-bit.

MOD376 or MOD386 (continued)

NOTES

Do not specify a USE16, USEREAL, or VIRTUALMODE attribute in the build program with MOD376 in effect.

Do not specify a PAGING definition in the build program with MOD376 in effect.

Do not specify the PAGETABLES control with MOD376.

Examples

1. In this example, a bootloadable file named MOD is created from linkable modules in the input file MOD.OBJ, which is executable on an 80376.

DOS: BLD386 MOD.OBJ OBJECT (MOD) BOOTLOAD MOD376

VMS: BLD386/OBJECT=MOD/BOOTLOAD/MOD376 MOD.OBJ

2. In this example, a bootloadable file MOD is created which is executable on an 80386.

DOS: BLD386 MOD.OBJ OBJECT (MOD) BOOTLOAD MOD386

VMS: BLD386/OBJECT=MOD/BOOTLOAD/MOD386 MOD.OBJ

OBJECT

Designates or suppresses
object module

Syntax

DOS: OBJECT [(*filename*)]
NOOBJECT

VMS: /OBJECT [= *filename*]
/NOOBJECT

Abbreviation

OJ
NOOJ

Default

OBJECT using the first input file name with extension truncated for DOS, or with the extension .DAT for VMS.

Description

OBJECT names the file containing a loadable or bootloadable output module. If *filename* is specified in the control, BLD386 assigns the specified name to the output file. If neither OBJECT nor NOOBJECT is specified, or if OBJECT is specified without *filename*, BLD386 assigns a default file name. The default file name is the name of the first file in the input list with the extension truncated for DOS, or with the extension .DAT for VMS.

The NOOBJECT control suppresses the creation of a loadable or bootloadable object module.

NOTE

If the default or specified *filename* matches any output file, or a file in the input list, or a file in a control file, BLD386 processing aborts.

OBJECT (continued)

Example

In this example, BLD386 uses input from MOD.OBJ to produce a loadable object module that is placed in the output object file MOD.

DOS: BLD386 MOD.OBJ OBJECT (MOD) NOBOOTLOAD

VMS: BLD386/NOBOOTLOAD/OBJECT=MOD MOD.OBJ



Syntax

DOS: PAGETABLES

VMS: /PAGETABLES


Abbreviation

PT

Default

No page tables are generated.

Description




PAGETABLES specified with the BOOTLOAD control directs BLD386 to generate page tables.

BLD386 generates page tables only for memory specified with the PAGING definition or implicitly specified in the MEMORY definition. See Chapter 3 for information on these definitions.

NOTE

PAGETABLES is invalid with the MOD376 control.

Example



In this example, PAGETABLES is valid only if the BOOTLOAD control is specified either explicitly (as in this example) or implicitly by default.

DOS: BLD386 MOD.OBJ PAGETABLES BOOTLOAD

VMS: BLD386/PAGETABLES/BOOTLOAD MOD.OBJ

Invocation

PRINT

Designates or suppresses
print file

Syntax

DOS: PRINT [(*filename*)]
NOPRINT

VMS: /PRINT [= *filename*]
/NOPRINT

Abbreviation

PR
NOPR

Default

PRINT using the output object file name with .MP2 extension if the OBJECT control is in effect, or using the first input file name with .MP2 extension if the NOOBJECT control is specified.

Description

PRINT names the print file that, by default, contains a build program listing, maps of segments, gates, page tables, and tasks, plus warning and error messages.

If *filename* is specified, BLD386 assigns that name to the print file. If neither PRINT nor NOPRINT is specified, or if PRINT is specified without a *filename*, BLD386 assigns a default file name. The print file name is the same as that of the output object file with extension .MP2.

If the NOOBJECT control is in effect, the print file is assigned the name of the first input file with extension .MP2.

NOPRINT suppresses print file generation.

The contents and format of the print file are described in Chapter 5.

PRINT (continued)

See the **ERRORPRINT** control in this chapter, which creates a separate print file for error messages.

See the **[NO]WARNINGS** control in this chapter, which selectively suppresses messages in the print file.

NOTES

Even if **NOPRINT** is in effect and a fatal error condition occurs, fatal error messages are displayed at the console.

If the default or specified *filename* matches any file name in the input list, **BLD386** processing aborts.

Example

In this example, **BLD386** places all maps, all listings of the build program, and all warning and error messages in the file **MOD.LIS**.

DOS: **BLD386 MOD.OBJ PRINT (MOD.LIS) BUILDFILE (MOD.BLD)**

VMS: **BLD386/PRINT=MOD.LIS/BUILDFILE=MOD.BLD MOD.OBJ**

RELDESC

Outputs relocation
information

Syntax

DOS: RELDESC

VMS: /RELDESC

Abbreviation

RD

Default

No relocation information is generated.

Description

RELDESC directs BLD386 to output relocation information for GDT and LDT descriptors. This information can be used by loaders to relocate descriptors already installed in the GDT and LDT and to adjust the references to those descriptors in the output module.

RELDESC is effective only for loadable object modules.

Example

In this example, BLD386 creates a loadable module and generates relocation information.

DOS: BLD386 MOD.OBJ NOBOOTLOAD RELDESC

VMS: BLD386/NOBOOTLOAD/RELDESC MOD.OBJ

Syntax

DOS: **TITLE** (*string*)

VMS: **/TITLE** = *string*

Abbreviation

TT

Default

TITLE with the null string.

Description

TITLE directs BLD386 to place a string in the heading line at the top of each print file page as follows:

```
80386 SYSTEM BUILDER      string  
                        dd/mm/yy hh:mm:ss  PAGE   number
```

The string is truncated on the right, if necessary, to print the entire heading line.

If the string contains any characters defined as special by the operating system, the entire string must be delimited. For DOS, use apostrophes or double quotation marks as delimiters. For VMS, use double quotation marks.

TITLE (continued)

Example

In this example, BLD386 places the string THIS IS A TITLE as part of the title on each page of the print file.

DOS: BLD386 MOD.OBJ PRINT (MOD.LIS) TITLE ('THIS IS A TITLE')

VMS: BLD386/PRINT=MOD.LIS/TITLE="THIS IS A TITLE"

TYPE

Enables or suppresses
type checking

Syntax

DOS: TYPE
NOTYPE

VMS: /TYPE
/NOTYPE

Abbreviation

TY
NOTY

Default

TYPE

Description

TYPE directs BLD386 to perform type checking across public and external symbols with the same name and across external symbols with the same name.

NOTYPE suppresses type checking and causes BLD386 to omit type definitions.

Example

In this example, BLD386 performs type checking between public and external symbols with the same name found in the input files MOD1.OBJ and MOD2.OBJ.

DOS: BLD386 MOD1.OBJ, MOD2.OBJ TYPE

VMS: BLD386/TYPE MOD1.OBJ, MOD2.OBJ

Invocation

WARNINGS

Includes or suppresses
error and warning messages

Syntax

DOS: WARNINGS
NOWARNINGS [(*n* [,...])]

VMS: /WARNINGS
/NOWARNINGS [= (*n* [,...])]

Abbreviation

WA
NOWA

Default

WARNINGS

Description

Use [NO]WARNINGS to control the output of warning and error messages in the print file (see Chapter 5), in the file created with the ERRORPRINT control, and at the console.

Unless NOWARNINGS is specified, BLD386 does not suppress any warnings or errors.

If NOWARNINGS is specified without any parameters, only warnings (not error messages) are suppressed.

If *n* is specified with NOWARNINGS, BLD386 suppresses only those errors or warnings corresponding to the numbers in the list (except for fatal errors). See Appendix C for the error and warning messages and their numbers.

Example

In this example, BLD386 produces a bootloadable object module MOD.X from the two linkable files MOD.LNK and CPROG.OBJ with warnings 244 and 247 suppressed.

DOS: BLD386 MOD.LNK, CPROG.OBJ OBJECT (MOD.X) BOOTLOAD NOWARNINGS (244,247)

VMS: BLD386/OBJECT=MOD.X/BOOTLOAD/NOWARNINGS=(244,247) MOD.LNK,CPROG.OBJ



Ordering and shipping

12

Contents

Chapter 5 Input and Output

5.1	File Types	5-1
5.2	Print File.....	5-3
5.2.1	Header	5-3
5.2.2	Build Program Listing	5-4
5.2.3	Segment Map.....	5-4
5.2.4	Gate Table.....	5-7
5.2.5	Task Table.....	5-8
5.2.6	Page Tables	5-12
5.3	Warning and Error Messages	5-13



This chapter describes the kinds of input and output the 80386 System Builder accepts. Section 5.2 details the output print file.

5.1 File Types

BLD386 accepts the following as input:

- a build program file
- linkable-module output from BND286, BND386, 80286 translators, or 80386 translators
- linkable modules from libraries of 80286 or 80386 object modules
- modules exported from BLD286 or BLD386

Figure 5-1 shows the kinds of input and output. Specify BLD386 controls and input and output files in the invocation line, in control files, or both.

BLD386 creates a loadable or bootloadable module as output, and optionally creates one or more linkable modules containing exported entities. If specified, the print file contains a build program listing; a map of segments, gates, and tasks in the output object module; and warning and error messages.

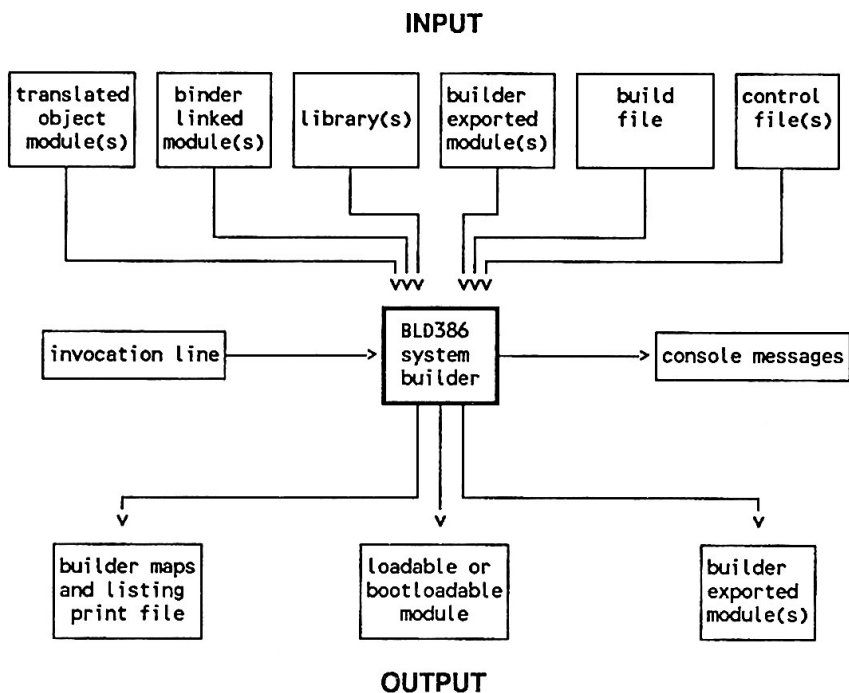


Figure 5-1 BLD386 Input and Output

If output file names are not specified in the controls, BLD386 assigns them by default as follows:

- The file containing the loadable or bootloadable module has the same file name as the first input file name, with the extension truncated for DOS, or with the .DAT extension for VMS.
- The print file has the same file name as that of the output file, with a new extension of .MP2.

NOTE

If any of the file names match (input or output, default or specified), BLD386 aborts processing and issues an error message.

5.2 Print File

The print file contains the following sections in order:

- Header
- Build program listing (if the LIST control is in effect)
- Segment map (if the MAP control is in effect)
- Gate table (if the MAP control is in effect)
- Task table (if the MAP control is in effect)
- Page directory (if the MAP, PAGETABLES, and MOD386 controls are in effect)
- Page tables (if the MAP, PAGETABLES, and MOD386 controls are in effect)

Any error and warning messages are throughout the listing. The following explanations of the print file sections include generic illustrations of the print file format.

5.2.1 Header

The print file header (see Figure 5-2) summarizes the invocation conditions.

```
80386 SYSTEM BUILDER      title
                        dd/mm/yy hh:mm:ss  PAGE      number

system-id 80386 SYSTEM BUILDER, Vx.y

INPUT FILES: filename ,...
OUTPUT FILE: filename
CONTROLS SPECIFIED: control ...
```

Figure 5-2 BLD386 Print File Header

5.2.2 Build Program Listing

The print file includes the build program listing if the LIST control is in effect (see Figure 5-3) during the build. BLD386 refers to the numbers on the build program lines for error reporting. If an error is associated with a build program line, BLD386 inserts a message after the line. The text of the message contains a reference to the line number, as described in Appendix C.

BUILD FILE: filename

```
1  xxxxxx;  
2  SEGMENT xxxxxx(LEVEL2), ...  
3  .  
.  
.  
16 TASK ...;  
17 TABLE ...;  
18 END
```

BUILD FILE PROCESSING COMPLETED

Figure 5-3 BLD386 Build Program Listing

The left-hand column in Figure 5-3 indicates the build program line number for error reporting. The last line indicates that build program processing has finished.

5.2.3 Segment Map

BLD386 creates the segment map if the MAP control is in effect (see Figure 5-4). It contains descriptor information for each input segment except Task State Segments (TSSs). TSSs are detailed in the task table. Segments of length 0 are omitted from the segment map.

SEGMENT MAP

TABLE	PBIT	DPL	ACCESS	USE	BASE	LIMIT	SEGMENT NAME
-------	------	-----	--------	-----	------	-------	--------------

GDT

xxxx	x	x	xxxx	xx	xxxxxxxxxH	xxxxxxxxxH	xxxxxxxxxxxxx
xxxx	x	x	xxxx	xx	xxxxxxxxxH	xxxxxxxxxH	xxxxxxxxxxxxx
.
.

LDT.1 (ttt)

xxxx	x	x	xxxx	xx	xxxxxxxxxH	xxxxxxxxxH	xxxxxxxxxxxxx
.
.

Figure 5-4 BLD386 Print File Segment Map

BLD386 provides the following information for each segment in the output object module:

TABLE

shows the location of a segment in a descriptor table. For each table, the index values for the segment descriptors are listed under a table identifier as follows:

GDT and IDT identify the global descriptor table and the interrupt descriptor table.

LDT.*n* (*ttt*) identifies each local descriptor table, where *n* is assigned by BLD386 according to the definition order of the LDTs in the build program, and *ttt* is the name of the LDT. BLD386 assigns LDT? as the default LDT name.

PBIT

is the setting of the present bit in the descriptor's access byte:

- 1 = segment is present in physical memory.
- 0 = segment is not present in physical memory.

DPL	is the segment descriptors privilege level, a value from 0 through 3.
ACCESS	<p>is one or more of the following segment attributes:</p> <ul style="list-style-type: none"> • C Conforming. Programs that execute at lower (numerically higher) privilege levels can access this code segment without using gates. • D ExpandDown. This is a nonexecutable segment whose limit can be extended toward low-order addresses at run time. • EO Execute Only. This is an execute-only code segment. • ER Execute and Read. This is an executable and readable code segment. • RO Read Only. This is a read-only data segment. • RW Readable and Writable. This is a readable and writable data segment.
USE	is the B/D-bit which can be set to USE16, USE32, or USEREAL. The setting is 0 for 8086 and 80286, and 1 for 80386. The segment map shows "16" in the case of USE16, "32" for USE32, or "RL" if USEREAL is specified.
BASE	is the hexadecimal equivalent of any absolute address assigned to be the segment's base. The base value can be adjusted to reflect address conventions for expanddown segments. For USE16 segments, the actual physical base (lowest address) = the base in the descriptor + the limit in the descriptor + 2. The size of a USE16 segment = 64K bytes - the limit in the descriptor - 1. For USE32, the stack size must be a multiple of one page (4K bytes).

LIMIT

is the hexadecimal segment limit in bytes. The limit is the offset of the last byte in the segment, counting from 0. The value can be influenced by the following conditions:

- The limit can be a few bytes longer to accommodate references to the last byte of the segment.
- The limit can be adjusted to reflect address conventions for expanddown segments.

SEGMENT NAME

is the name of the segment.

GDT or IDT are for the aliases of the GDT or the IDT in bootloadable modules.

/// is for an LDT defined and named in the build program.

LDT? is for an automatically created LDT.

5.2.4 Gate Table

Figure 5-5 shows the gate table section of the print file. BLD386 produces the gate table if the MAP control is in effect. The gate table contains information for every gate in the output object modules. Uninitialized fields are filled with dashes.

GATE TABLE

GATE NAME	TABLE	PBIT	DPL	TYPE	WC	SELECTOR	OFFSET
xxxxxxxxxx	GDT(xxxx)	x	x	x86xxxx	xx	GDT(xxxx)	xxxxxxxxxH
xxxxxxxxxx	LDT.1(xxxx)	x	x	x86xxxx	xx	GDT(xxxx)	xxxxxxxxxH
.
.
.
xxxxxxxxxx	LDT.1(xxxx)	x	x	x86xxxx	xx	GDT(xxxx)	xxxxxxxxxH

Figure 5-5 BLD386 Print File Gate Table

GATE NAME	is the name of the gate. Gates are listed in alphabetical order by gate name.
TABLE	is a table designation and index (in parentheses) of where the gate descriptor is placed. The table designation is GDT, IDT, or LDT. <i>n</i> . The LDT name which corresponds to an LDT. <i>n</i> is the name shown in the Segment Map. BLD386 assigns <i>n</i> according to the definition order of the LDTs in the build program.
PBIT	is the setting of the present bit in the gate: <ul style="list-style-type: none"> • 1 = descriptor contents are valid. • 0 = descriptor contents are not valid.
DPL	is the gate's privilege level, a value from 0 through 3.
TYPE	denotes the gate type. The seven gate types are represented as follows: <p style="margin-left: 40px;">286CALL or 386CALL TASK 286INTR or 386INTR 286TRAP or 386TRAP</p>
WC	is the word count for call gates.
SELECTOR	is a table designation and a parenthesized index where the descriptor for the target segment is placed. GDT and IDT describe the global descriptor table and interrupt descriptor table. LDT. <i>n</i> corresponds to the name shown in the Segment Map.
OFFSET	is the hexadecimal representation of the segment offset of the entry point for call, interrupt, and trap gates.

5.2.5 Task Table

BLD386 produces the task table (see Figure 5-6) if the MAP control is in effect. Tasks are listed in alphabetical order. BLD386 fills uninitialized fields with dashes.

TASK TABLE

```
ttttttttt: TABLE = GDT(xxxx)  PBIT = x  DPL = x
      BASE = xxxxxxxxH
      LIMIT = xxxxH
      SS0:ESP0= LDT.1(xxxx):xxxxxxxxxH
      SS1:ESP1= LDT.1(xxxx):xxxxxxxxxH
      SS2:ESP2= LDT.1(xxxx):xxxxxxxxxH
      SS:ESP = LDT.1(xxxx):xxxxxxxxxH
      CS:EIP = LDT.1(xxxx):xxxxxxxxxH
      PDR = xxxxxxxxH
      DS = LDT.1(xxxx)
      LDT = xxxxxxxx AT GDT(xxxx)
      IOPRIV = xxH
      INTERRUPT [NOT] ENABLED
      DEBUG [NOT] ENABLED
      VIRTUALMODE
```

.
.
.

```
ttttttttt: TABLE = GDT(xxxx)  PBIT = x  DPL = x
      BASE = xxxxxxxxH
      LIMIT = xxxxH
      SS0:ESP0= LDT.1(xxxx):xxxxxxxxxH
      SS1:ESP1= LDT.1(xxxx):xxxxxxxxxH
      SS2:ESP2= LDT.1(xxxx):xxxxxxxxxH
      SS:ESP = LDT.1(xxxx):xxxxxxxxxH
      CS:EIP = LDT.1(xxxx):xxxxxxxxxH
      PDR = xxxxxxxxH
      DS = LDT.1(xxxx)
      LDT = xxxxxxxx AT GDT(xxxx)
      IOPRIV = xxH
      INTERRUPT [NOT] ENABLED
      DEBUG [NOT] ENABLED
      INITIAL
```

Figure 5-6 BLD386 Print File Task Table

<i>TTTTTTTT</i>	is the task's name. BLD386 assigns TASK? as the default task name.
TABLE	is the location of the TSS descriptor, expressed as GDT(<i>index</i>).
PBIT	is the setting of the present bit in the TSS descriptor: <ul style="list-style-type: none"> • 1 = descriptor contents are valid. • 0 = descriptor contents are not valid.
DPL	is the TSS descriptor privilege level, a value from 0 through 3.
BASE	is the hexadecimal equivalent of the 32-bit absolute base address assigned to the TSS.
LIMIT	the hexadecimal value that represents the offset of the last byte in the TSS.
SS0:ESP0	is the initial register value for the level 0 stack.
SS1:ESP1	is the initial register value for the level 1 stack.
SS2:ESP2	is the initial register value for the level 2 stack.
SS:ESP	is the initial register value for the stack pointer.
PDR	is the absolute base address of the page table directory.
CS:EIP	is the entry point of the initial code segment.
DS	is the initial data segment.

Register initialization values are expressed as *table-designation(index) : 32-bit offset* which represents *segment descriptor location : segment offset*. For an 8086 VIRTUALMODE task, the value represents *paragraph number : offset*.

The table designation is GDT, IDT, or LDT.*n*, where LDT.*n* is the alias name of the segment (see the segment name column of the segment map in Figure 5-4). The name that corresponds to LDT.*n* is described in Section 5.2.3. The index is in decimal notation.

PDR	is the page table directory base. This is the absolute base address of the page table directory if paging has been specified.
LDT	is the LDT name that has a selector installed in the TSS, and the GDT index where the descriptor for the LDT resides.
IOPRIV	is the setting of the I/O privilege bits in the flag word of the TSS.
INTERRUPT	is the setting of the interrupt flag (IF) in the flag word of the TSS. If IF = 1 then the line is printed as: INTERRUPT ENABLED If IF = 0 then the line is printed as: INTERRUPT NOT ENABLED
DEBUG	is the setting of the debug trap flag (T) in the TSS. If T = 1 then an exception is raised on a task switch, and the line is printed as: DEBUG ENABLED If T = 0 then the line is printed as: DEBUG NOT ENABLED
INITIAL	only appears with the TSS for the initial task to be run.
VIRTUALMODE	is the setting of the VM flag in the EFLAGS register. This line only appears if the task is defined as VIRTUALMODE in the TASK definition of the build program. In this case, all selector values in the register initialization fields are paragraph numbers, in order to access 8086 addresses.

5.2.6 Page Tables

Figure 5-7 shows the page tables section of the print file. One map is for the page directory, and one is for the page tables. BLD386 produces this section if the MAP control is in effect. BLD386 produces page tables and a page table directory for bootloadable output only if the PAGETABLES and MOD386 controls are in effect.

```
PAGE DIRECTORY      xxxxxxxxH

LINEAR ADDRESS
FROM      TO      TABLE      P RW US  UD1 UD2 UD3
xxxxxxx .. xxxxxxx xxxxxxxxH  x x x  x  x  x
xxxxxxx .. xxxxxxx -----  x x x  x  x  x
.
.

PAGE TABLES

LINEAR ADDRESS
FROM      TO      P RW US  UD1 UD2 UD3
xxxxxxx .. xxxxxxx  x x x  x  x  x
xxxxxxx .. xxxxxxx  x x x  x  x  x
.
.
```

Figure 5-7 BLD386 Page Tables

PAGE DIRECTORY is the absolute base address of the directory.

LINEAR ADDRESS is the base and limit of the corresponding page table for each 4M bytes of memory, if some or all of the memory is marked as present.

P is the present bit for the page table.

RW is the read/write bit for the page table.

US is the user/supervisor bit for the page table.

UD1, UD2, UD3 are the user-defined bits for the page table.

PAGE TABLES lists the pages.

LINEAR ADDRESS	is the base and limit of the first page corresponding to this memory range.
P	is the present bit for the page.
RW	is the read/write bit for the page.
US	is the user/supervisor bit for the page.
UD1, UD2, UD3	are the user-defined bits for the page.

5.3 Warning and Error Messages

Appendix C defines the warning, error, and fatal error messages that can appear in the print file. Messages associated with the build program appear with the build program listing. Other messages appear at the end of the print file. Warnings and non-fatal error messages can be suppressed in the print file (see [NO]WARNINGS control in Chapter 4).






Contents

Chapter 6 System Building Examples

6.1	Flat Model Templates for DOS.....	1
6.1.1	The Flat Memory Model.....	2
6.1.2	The Flat Model Templates.....	3
6.1.2.1	The Initialization Template.....	3
6.1.2.2	The Build Program Template.....	9
6.1.2.3	The Interrupt Routines.....	13
6.1.2.4	The C Startup Template.....	14
6.1.2.5	The DOS Batch File	16
6.1.3	The Print File	18
6.2	Using 80386 Call Gates with C.....	21
6.2.1	The Call and the Return	21
6.2.2	Stack Cleanup.....	22
6.2.2.1	Example Interface Routines.....	23
6.2.2.2	ASM386 Startup Code	25
6.2.2.3	The Build Program	26
6.2.2.4	The DOS Batch File	27






This chapter provides in-depth examples of system building using BLD386.

6.1 Flat Model Templates for DOS


Creating a flat, or linear, model bootloadable system for an Intel386™ family microprocessor is straightforward. BLD386 assigns absolute addresses and makes the necessary system data structures.

The 80386 gives 86-family programmers the ability to program in a linear address space. For systems requiring large code/data space, the flat memory model is easy to use. This section describes templates created for programming the 80386 (and the 80376) using a flat memory model.



Initializing the processor to run in 32-bit protected mode must be done after system reset. One template is designed to initialize either the 80386 or the 80376 to run in protected mode. The key to this initialization template is the use of BLD386, the builder, to create the absolute protected-mode system.

BLD386 builds absolute protected-mode systems from the input segments provided by the user. System implementation details are input to BLD386 by way of a build program. The build program contains information such as segment protection levels, descriptor table definitions, task creation details, and memory configuration. There is a template build program used to create a flat protected-mode system.



6.1.1 The Flat Memory Model

Figure 6-1 depicts the flat memory model used by the builder.

The feature of the builder that makes the flat memory model easy to use is the FLAT control. When FLAT is used, BLD386:

- Creates two segments called `_phantom_code_` and `_phantom_data_`. The default DPL (descriptor privilege level) value is zero. Both these segments have a default base address of zero and a default limit of 4G bytes - 1.
- Combines input segments into the corresponding phantom segment in the order of input. BLD386 does not overlap true code and data; when ROM areas and RAM areas are specified, BLD386 allocates space for segments according to their access attributes: read-only and execute-only go to ROM, read/write goes to RAM.

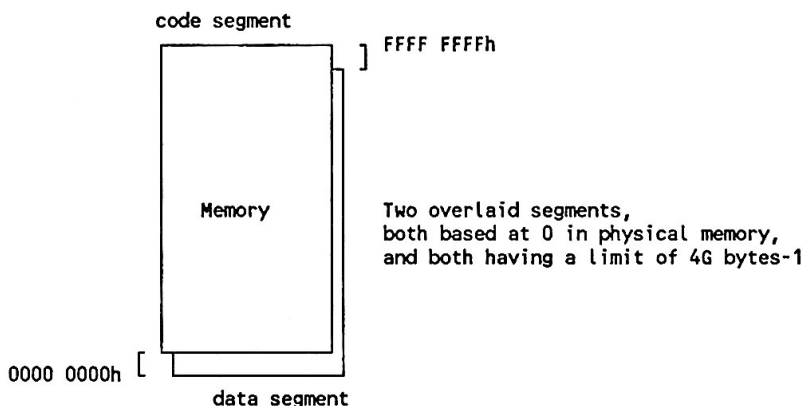


Figure 6-1 Builder's 80386 Flat Memory Model

- Adjusts gates and all relative and absolute jumps to use the base of the `_phantom_code_` selector.
- Adjusts all data and stack accesses to use the base of the `_phantom_data_` selector.

By using the FLAT control, you can implement many details of a flat model system automatically.

6.1.2 The Flat Model Templates

All of these templates are created so that they can be used in many different systems and are easy to understand. They are at the heart of a typical embedded, protected-mode system.

6.1.2.1 The Initialization Template

The initialization code template `flat.a38` contains two distinct modules. The first module, `INIT_CODE`, simply places the processor into 32-bit protected mode. Since the code runs on both the 80386 (which, at RESET, is in 16-bit real mode) and the 80376 (which, at RESET, is in 32-bit mode), the `INIT_CODE` segment is a 32-bit segment. For correct 80386 execution, however, segment overrides force the processor to execute the 32-bit form of various instructions.

The other startup module, `COPYTABLES`, copies part of the GDT (global descriptor table) created by the builder from ROM to RAM. In this example, the RAM location of the GDT is calculated by ANDing the ROM location with `0FFFF0000h`. The code only copies five GDT descriptor entries down into RAM. `COPYTABLES` also copies part of the IDT (interrupt descriptor table) from ROM to RAM. This example has only 16 true IDT entries.

To determine the RAM location of the descriptor tables, both startup modules use the GDT and IDT aliases that `BLD386` creates and places into the GDT by default. Because these aliases already exist, it is easy to modify the address and size of system descriptor tables.

If you are implementing a ROM-based system, then care must be taken when using this startup code. The microprocessor attempts to write to the GDT descriptor's `ACCESS` bit upon execution of an `LGDT` instruction. Since the GDT descriptor `GDT_DESC` is in ROM, your hardware must return a `READY` upon a write to ROM. If your hardware does indeed return a `READY` after a write to ROM, then the step of copying descriptor tables down to RAM in the `COPYTABLES` module may be skipped: simply replace the far jump destination at the end of the `INIT_CODE` module with that of your code.

```

; flat.a38
; Startup code for flat (linear) model example
;
; *****
;
; Version 1.1
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors. Intel hereby grants
; you permission to modify and incorporate it as needed.
;
; *****
;
;
; This is an example of startup code to put the 80386/80376 into flat mode.
; It should work with all applications, but there are certain assumptions
; made in this model. The memory model used is a typical embedded
; application model. Descriptor tables and code reside in ROM and data
; is assumed to be in RAM. This example assumes that ROM begins at
; 0FFFF0000h; since descriptor tables need to be RAM-based for protected-
; mode execution, the code copies the builder-created GDT and IDT down into
; RAM with the RAM address being <ROM_address AND 0FFFF0000h>. It also
; assumes a very simple model with only five GDT entries: NULL, a
; GDT alias, an IDT alias, CODE, and DATA. The builder creates the GDT alias
; and IDT alias and places them, by default, in GDT[1] and GDT[2].
; After entering protected mode, this code jumps to a C startup routine.
; You may change this jmp address to that of your code, or make the label of
; your code "c_startup".

```

```

NAME FLATSTART          ; name of object module

```

```

EXTRN c_startup:near    ; this is the label jumped to after init and copytables

```

```

pe_flag      equ 1
data_selc    equ 20h    ; assume code is GDT[3], data GDT[4]
galias_off   equ 8      ; offset of GDT alias in GDT
ialias_off   equ 10h    ; offset of IDT alias in GDT
idt_lim      equ 80h    ; assume that 16 entries are all that are needed in IDT
gdt_lim      equ 2ch    ; assume that 5 entries are all that are needed in GDT

```

```

CODEMACRO      startup PREFIX
                db 66h
ENDM

```

```

INIT_CODE      SEGMENT ER PUBLIC USE32

```

```

; GDT_DESC and IDT_DESC are public symbols used in the build file.
; The LOCATION definitions in the TABLE section of the build file point to
; these labels; the builder will store the descriptor for the named table
; at this location in memory.

```

```

PUBLIC         GDT_DESC
PUBLIC         IDT_DESC

```

```

gdt_desc       dq ?
idt_desc       dq ?

```

```

; START is a label that points to the true beginning of our executable
; code. The BOOTSTRAP option of the builder invocation line causes
; a short jump to the named label (in this case, START) to be placed at
; the component reset vector.

```

```

PUBLIC         START

```

```

; Since this code is designed to initialize either an 80386 or an 80376 into
; protected mode, the first two instructions test for component-type.
; The 80386 at reset is in real or compatibility mode: the PE bit is
; off and the D bit for CS is not set. This causes instructions to be
; executed in their 16-bit form. The 80376 at reset has the PE bit on as
; well as the D bit, so instructions are executed in their 32-bit form.

```

```

        nop                ; remove the nop's if initializing an 80386
        nop
start:
        cld                ; clear direction flag
        smsw bx            ; check for processor (80376) at reset
        test bl,1          ; use SMSW rather than MOV for speed
        jnz pestart

```

```

realstart:                ; is an 80386 and in real mode
                          ; macro forces next operand into 32-bit mode
startup mov eax,offset gdt_desc
                          ; move address of the GDT descriptor into eax
xor ebx,ebx               ; clear ebx
mov bh,ah                 ; load 8 bits of address into bh
mov bl,al                 ; load 8 bits of address into bl
                          ; use the 32-bit form of LGDT to load
lgdtw cs:[ebx]            ; the 32 bits of address into the GDTR

; If gdt_desc is in ROM, then user
; hardware MUST return a ready after
; a write.

```

```

smsw ax                   ; go into protected mode (set PE bit)
or al,pe_flag
lmsw ax
jmp next                  ; flush prefetch queue

```

```

pestart:
mov eax,offset gdt_desc
lgdt [eax]
xor ebx,ebx               ; initialize data selectors
mov bl,data_selc         ; GDT[3]
mov ds,bx
mov ss,bx
mov es,bx
mov fs,bx
mov gs,bx
jmp pejump

```

```

next:
xor ebx,ebx               ; initialize data selectors
mov bl,data_selc         ; GDT[3]
mov ds,bx
mov ss,bx
mov es,bx
mov fs,bx
mov gs,bx

```

```
pejump:
    startup jmp far ptr copytables    ; the 80376 is already in 32-bit mode
```

```
INIT_CODE ENDS
```

```
////////////////////////////////////
```

```
CODE32 SEGMENT ER PUBLIC USE32
```

```
copytables:
```

```
; Copy GDT and IDT from ROM to RAM.
; Assume the RAM location for the tables
; is the ROM location AND 0FFFF0000h. This
; code thinks that a GDT alias is the third
; GDT entry (GDT[2] or gdt_desc+8).
```

```
    mov eax, offset gdt_desc
    mov ebx, dword ptr [eax]+2 ; base of GDT
```

```
; Move the descriptors from GDT to RAM location = gdt_desc AND 0FFFFh.
```

```
    mov esi,ebx                ; GDT offset down in RAM area
    and ebx,0ffffh            ; new address of GDT-to-be
    mov edi,ebx                ; mov into destination register for string move
    mov ecx,gdt_lim            ; decrease the limit to only 4 entries
    rep movsb                  ; move 4 descriptors from ROM to RAM
```

```
; Here we want to modify the GDT alias at GDT[1] so we can re-load
; the GDT with the new RAM-based value. Why modify? The builder
; places a DATA segment descriptor (read/write) into this location so that the
; GDT can be modified. Since we want to load this descriptor into
; the GDTR, we need to change it to be like gdt_desc. We copy the
; gdt_desc contents down from ROM, change the base and limits to our current
; values, load the GDTR, then change the GDT alias back to its original
; value.
```

```

mov esi,eax                ; eax still holds offset of gdt_desc
add esi,4                  ; 4 = offset of second dword of gdt_desc
mov edi,ebx                ; ebx = new base address of GDT while
add edi,galias_off+4       ; 4 = offset of second dword of GDT alias
mov edx,[edi]              ; store second dword of GDT alias into EDX
and edx,0ffff00h          ; change base to new base

movsd                      ; move 8-byte descriptor into GDT alias slot
mov word ptr [ebx]+ galias_off,gdt_lim ; change limit to 4 descs
mov dword ptr [ebx]+ galias_off+2,ebx ; move new address into GDT alias
lgdt word ptr [bx] + galias_off      ; load GDT alias into GDTR
mov dword ptr [ebx]+ galias_off+4,edx ; restore the GDT alias desc

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; IDT mov from ROM to RAM

mov eax, offset idt_desc
mov ebx, dword ptr [eax]+2 ; base of IDT

; Move the descriptors from IDT to RAM location = idt_desc AND 0FFFFh.

mov esi,ebx                ; IDT offset down in RAM area
and ebx,0ffffh             ; new address of IDT-to-be
mov edi,ebx
mov ecx,idt_lim            ; this is to move only 16 entries
rep movsb                  ; move 16 descriptors from ROM to RAM

; Change descriptor for the IDT to point to the new location
; GDT[2], or (base of GDT + ialias_off), is the
; alias for the IDT descriptor. EBX holds new base of IDT.

mov eax,offset gdt_desc    ; need to find where GDT is now
mov edi, dword ptr [eax]+2 ; base of GDT
and edi,0ffffh            ; new location of GDT
add edi,ialias_off         ; offset of IDT alias
mov edx,[edi+4]            ; store second dword of IDT alias into EDX
and edx,0ffff00h          ; change base to new base

```



```

mov [edi+4],ebx          ; mov new base into IDT alias
mov dword ptr [edi],idt_lim ; change limit to 16 descs
lidt word ptr [edi]      ; load IDT alias into IDTR
mov dword ptr [edi+4],edx ; restore the IDT alias desc

jmp c_startup           ; jump to user program

```

CODE32 ENDS

END

6.1.2.2 The Build Program Template

The `flat.bld` build file containing the build program is both simple and generically useful. The following explains what each specification means and why it is used.

- SEGMENT** This definition sets the DPL of all input segments to zero. The builder creates the `__phantom_code__` and `__phantom_data__` segments when the FLAT control is used; they are included here as a reminder even though their default DPL is zero.
- TABLE** This defines the GDT.
- LOCATION** This specification places the descriptor describing the GDT into memory defined in the user program: the symbol `GDT_DESC` is an uninitialized 6-byte area in the `INIT_CODE` module. This feature is handy when you are relocating code often: since BLD386 "fixes up" the absolute value stored in memory, you can change the location either of your initialization module or of your GDT without re-compiling any code.
- BASE** This specification absolutely locates the GDT at the specified spot in memory. The GDT goes in ROM. This specifies the exact base address of this table.

TASK	This definition defines a task so that the debugger initializes the microprocessor's state upon a LOAD into RAM. If a TSS is present, then the ICE TM -386 TM In-Circuit Emulator places the microprocessor in protected mode and initializes all the segment registers. A flat model system does not require a task, however.
GATE	The 80386 and the 80376 require that IDT entries be interrupt gates. Instead of creating gate descriptors in assembly language, BLD386 creates them with the GATE definition. The ENTRY specification for each GATE specifies the public label of the interrupt handlers.
TABLE	The interrupt gates are placed in the IDT.
MEMORY	This definition describes the physical memory setup of the hardware, and defines the location of the software system.
RESERVE	BLD386 cannot place any code or data in these ranges. Use RESERVE to specify holes in the memory space, or to define areas used in other ways (like destinations of descriptor tables).
RANGE	This specifies ROM or RAM areas.

```
-- flat.bld
-- Build file for input to BLD386 to create flat model example
--
-- *****
--
-- Version 1.1
-- Copyright Intel Corp., 1988
-- This template is intended for your benefit in developing applications/
-- systems using Intel386(TM) family microprocessors. Intel hereby
-- grants you permission to modify and incorporate it as needed.
--
-- *****
--
```

FLAT; -- build program id

SEGMENT

```
*segments(dpl=0),          -- Give all user segments a DPL of 0.
_phantom_code_(dpl=0),      -- These two segments are created by
_phantom_data_(dpl=0);      -- the builder when the FLAT control is used.
```

TABLE

-- create GDT

GDT

```
(LOCATION = GDT_DESC,        -- In a buffer starting at GDT_DESC,
                                -- BLD386 places the GDT base and
                                -- GDT limit values. Buffer must be
                                -- 6 bytes long. The base and limit
                                -- values are places in this buffer
                                -- as two bytes of limit plus
                                -- four bytes of base in the format
                                -- required for use by LGDT and
                                -- LIDT instructions.
```

```
    BASE = 0ffff0100h
); -- end GDT
```

TASK

MAIN_TASK

```
(BASE = 0ffff0200h,
  DPL = 0,          -- Task privilege level is 0.
  DATA = DATA,    -- Points to a segment that
                    -- indicates initial DS value.
  CODE = main,       -- Entry point is main, which
                    -- must be a public id.
  STACKS = (DATA),   -- Segment id points to stack
                    -- segment. Sets the initial SS:ESP.
  NO INTENABLED,     -- Disable interrupts.
  PRESENT            -- Present bit in TSS set to 1.
);
```

GATE

```
INT0_GATE (INTERRUPT, DPL = 0, ENTRY = INT0),
INT1_GATE (INTERRUPT, DPL = 0, ENTRY = INT1),
INT2_GATE (INTERRUPT, DPL = 0, ENTRY = INT2),
INT3_GATE (INTERRUPT, DPL = 0, ENTRY = INT3),
INT4_GATE (INTERRUPT, DPL = 0, ENTRY = INT4),
INT5_GATE (INTERRUPT, DPL = 0, ENTRY = INT5),
INT6_GATE (INTERRUPT, DPL = 0, ENTRY = INT6),
INT7_GATE (INTERRUPT, DPL = 0, ENTRY = INT7),
INT8_GATE (INTERRUPT, DPL = 0, ENTRY = INT8),
INT9_GATE (INTERRUPT, DPL = 0, ENTRY = INT9),
INT10_GATE (INTERRUPT, DPL = 0, ENTRY = INT10),
INT11_GATE (INTERRUPT, DPL = 0, ENTRY = INT11),
INT12_GATE (INTERRUPT, DPL = 0, ENTRY = INT12),
INT13_GATE (INTERRUPT, DPL = 0, ENTRY = INT13),
INT14_GATE (INTERRUPT, DPL = 0, ENTRY = INT14),
INT16_GATE (INTERRUPT, DPL = 0, ENTRY = INT16);
```

TABLE

```
-- create IDT

IDT
  (LOCATION = IDT_DESC,
   BASE = 0ffff0000h,

   -- slots 0 through 31 are Intel reserved

   ENTRY = (0:INT0_gate, 1:INT1_gate, 2:INT2_gate, 3:INT3_gate,
            4:INT4_gate, 5:INT5_gate, 6:INT6_gate, 7:INT7_gate,
            8:INT8_gate, 9:INT9_gate, 10:INT10_gate, 11:INT11_gate,
            12:INT12_gate, 13:INT13_gate, 14:INT14_gate, 16:INT16_gate)
  ); -- end IDT
```

MEMORY

```
(RESERVE = (0..250h), -- for IDT and GDT
 RANGE = ( -- begin configuration section --
   ROM_AREA = ROM(0ffff0000h..0ffffff0h),
   RAM_AREA = RAM(251h..0ffffh)
 ) -- end configuration section --
);
```

END

6.1.2.3 The Interrupt Routines

A listing of the interrupt stub procedures follows. Intel has reserved the first 32 interrupts. Those which have been defined are included in the stub procedures.

```
; intrpt.a38
; Interrupt fault handlers for use with flat model example
;
; *****
;
; Version 1.1
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors. Intel hereby
; grants you permission to modify and incorporate it as needed.
;
; *****
```

NAME INTERRUPTS

intraput_routines SEGMENT ER PUBLIC

PUBLIC INT0,INT1,INT2,INT3,INT4,INT5,INT6,INT7,INT8,INT9,INT10,INT11

PUBLIC INT12,INT13,INT14,INT16

; FAULT-HANDLER: if an exception occurs, the corresponding fault handler is
; entered. The interrupt number is pushed on top of the stack.

```
INT0 : push 00h          ; divide error
      jmp ENTREZ
INT1 : push 01h          ; debug exceptions
      jmp ENTREZ
INT2 : push 02h          ; non-maskable interrupt
      jmp ENTREZ
INT3 : push 03h          ; breakpoint
      jmp ENTREZ
INT4 : push 04h          ; overflow
      jmp ENTREZ
INT5 : push 05h          ; bounds check
      jmp ENTREZ
```

```

INT6 :  push 06h          ; invalid opcode
        jmp ENTREZ
INT7 :  push 07h          ; coprocessor not available
        jmp ENTREZ
INT8 :  push 08h          ; double fault
        jmp ENTREZ
INT9 :  push 09h          ; coprocessor segment overrun
        jmp ENTREZ
INT10 : push 0ah          ; invalid tss
        jmp ENTREZ
INT11 : mov ax,ds         ; segment not present
        mov ds,ax         ; ensure full loading of the segment registers
        mov ax,es
        mov es,ax
        push 0bh
        jmp ENTREZ
INT12 : mov ax,ds         ; stack exception
        mov ds,ax         ; ensure full loading of the segment registers
        mov ax,es
        mov es,ax
        push 0ch
        jmp ENTREZ
INT13 : push 0dh          ; general protection
        jmp ENTREZ
INT14 : push 0eh          ; page fault
        jmp ENTREZ
INT16 : push 10h          ; coprocessor error

ENTREZ : hlt

inrupt_routines ENDS
END

```

6.1.2.4 The C Startup Template

The code in `cstart.a38` defines a stack for a C application. The stack pointer is initialized, and the C routine is called.

```

; cstart.a38
; An ASM386 module to initialize the stack and call a C application
;
; *****
;
; Version 1.1
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors. Intel hereby
; grants you permission to modify and incorporate it as needed.
;
; *****
;
name cstart                ; name of the object module
extrn main:near            ; label of the C application to be called
public c_startup          ; public symbol used in processor initialization code


data segment rw public

stack dd 1024 dup(?)      ; set up the stack
stacktop label dword
data ends


code32 segment er public

c_startup:
    mov esp,offset stacktop ; initialize stack pointer
    call main              ; call C application
    hlt                   ; halt processor

code32 ends

end

```

6.1.2.5 The DOS Batch File

A batch file called `flat.bat` assembles, compiles, binds, and builds the protected-mode system.

```
REM flat.bat
REM A DOS batch file for generating a bootloadable flat model example--
REM One input file as an argument is expected.
REM *****
REM
REM Version 1.1
REM Copyright Intel Corp., 1988
REM This template is intended for your benefit in developing applications/
REM systems using Intel386(TM) family microprocessors. Intel hereby
REM grants you permission to modify and incorporate it as needed.
REM
REM *****
echo off
REM
REM The following three invocations of ASM386 create object modules
REM "flat.obj", "intrpt.obj" and "cstart.obj". You will receive warnings with
REM each invocation due to use of privileged instructions in the files.
REM The "debug" control directs ASM386 to include extra information useful
REM in symbolic debugging. The listing files will be named "flat.lst",
REM "intrpt.lst", and "cstart.lst".
REM
asm386 flat.a38 debug
asm386 intrpt.a38 debug
asm386 cstart.a38 debug
REM
REM The invocation of C-386(TM) creates an object module "%1.obj", where the %1
REM is replaced with the name of the application file before the extension.
REM The "ra" control stands for "register allocate"; the compiler will
REM optimize the allocation of register variables. The "code" control
REM causes placement of a pseudo-assembly language listing at the end
REM of the listing file. "Debug" directs C-386 to include extra information
REM useful in symbolic debugging. The listing file will be named
REM "%1.lst".
REM
c386 %1.c debug ra code
REM
```


REM BND386 combines the input segments and resolves symbolic addressing.
REM The "nolo" control directs the binder to create a linkable (rather
REM than loadable) file. The "debug" control indicates that debug
REM information is to be retained. "Oj" directs the output file to be
REM named "%1.bnd". The listing file will be named "%1.mp1".

REM

bnd386 %1.obj,flat.obj,cstart.obj nolo debug oj (%1.bnd)

REM

REM Our goal is an absolute bootloadable file (all addresses fixed in memory)
REM suitable for loading into an ICE(TM)-386(TM). BLD386 creates such an
REM absolute module, necessary descriptor tables, and a task state segment for
REM initializing the ICE-386. The warnings pertaining to the phantom
REM segments' sizes being reduced should be ignored. The "bf" control
REM identifies "flat.bld" as the build file. The "bs" control identifies
REM the symbol "start" as the label of the instruction to be jumped to by
REM the bootstrap jump placed at 0ffffff0h. The "flat" control directs
REM the builder to configure the file in a flat, or linear, model (all
REM code will reside in the _phantom_code_ segment; all data will reside in
REM the _phantom_data_ segment). The "mod376" control causes the builder to
REM issue messages to guide creation of the object module for an 80376
REM processor. Remove this control to create an object module for an 80386.
REM The listing file will be named "%1.mp2".

REM

blt386 %1.bnd,intrpt.obj bf (flat.bld) bs (start) bl flat mod376

To create an example system, invoke the DOS batch file with the
name of a C-386TM application file.

flat filename.c

This line assembles, compiles, binds, and builds the system. The
assembler and BLD386 issue warning messages that can be ignored
(see the batch file itself for more information).

6.1.3 The Print File

The following is a part of the print file created when a simple C-386 application program called `newfib.c` was used as input to the DOS batch file above.

SEGMENT MAP

TABLE	PBIT	DPL	ACCESS	USE	BASE	LIMIT	SEGMENT NAME
GDT							
1	1	0	RW	16	FFFF0100H	0000004FH	GDT:
2	1	0	RW	16	FFFF0000H	000000F7H	IDT:
3	1	0	EO	32	00000000H	FFFFFFFFH	_PHANTOM_CODE_
4	1	0	RW	32	00000000H	FFFFFFFFH	_PHANTOM_DATA_
5	1	0	ER	32	FFFF0270H	000000F1H	NEWFIB.CODE32
6	1	0	RWD	32	00002260H	FFFFDFFFH	NEWFIB.DATA
7	1	0	ER	32	FFFF0150H	00000065H	NEWFIB.INIT_CODE
8	1	0	ER	32	FFFF0364H	00000051H	INTERRUPTS.INTRUPT_ROUTINES

GATE TABLE

GATE NAME	TABLE	PBIT	DPL	TYPE	WC	SELECTOR	OFFSET
INT1_GATE	IDT(1)	1	0	386INTR	0	GDT(3)	FFFF0368H
INT2_GATE	IDT(2)	1	0	386INTR	0	GDT(3)	FFFF036CH
INT3_GATE	IDT(3)	1	0	386INTR	0	GDT(3)	FFFF0370H
INT4_GATE	IDT(4)	1	0	386INTR	0	GDT(3)	FFFF0374H
INT5_GATE	IDT(5)	1	0	386INTR	0	GDT(3)	FFFF0378H
INT6_GATE	IDT(6)	1	0	386INTR	0	GDT(3)	FFFF037CH
INT7_GATE	IDT(7)	1	0	386INTR	0	GDT(3)	FFFF0380H
INT8_GATE	IDT(8)	1	0	386INTR	0	GDT(3)	FFFF0384H
INT9_GATE	IDT(9)	1	0	386INTR	0	GDT(3)	FFFF0388H
INT10_GATE	IDT(10)	1	0	386INTR	0	GDT(3)	FFFF038CH
INT11_GATE	IDT(11)	1	0	386INTR	0	GDT(3)	FFFF0390H
INT12_GATE	IDT(12)	1	0	386INTR	0	GDT(3)	FFFF039CH
INT13_GATE	IDT(13)	1	0	386INTR	0	GDT(3)	FFFF03A8H
INT14_GATE	IDT(14)	1	0	386INTR	0	GDT(3)	FFFF03ACH
INT16_GATE	IDT(16)	1	0	386INTR	0	GDT(3)	FFFF03B0H
INT0_GATE	IDT(0)	1	0	386INTR	0	GDT(3)	FFFF0364H

TASK TABLE

```
MAIN_TASK: TABLE = GDT(9)    PBIT = 1  DPL = 0
            BASE   = FFFF0200H
            LIMIT   = 00000067H
            SS0:ESP0= GDT(4):00002260H
            SS1:ESP1= -----
            SS2:ESP2= -----
            SS:ESP  = GDT(4):00002260H
            PDR     = -----
            CS:EIP  = GDT(3):FFFF0270H
            DS      = GDT(4)
            LDT     = -----
            IOPRIV  = 00H
            INTERRUPT NOT ENABLED
            DEBUG NOT ENABLED
            INITIAL
```

Figure 6-2 shows what the memory looks like with the location of the GDT and IDT after building the system with any small C application.

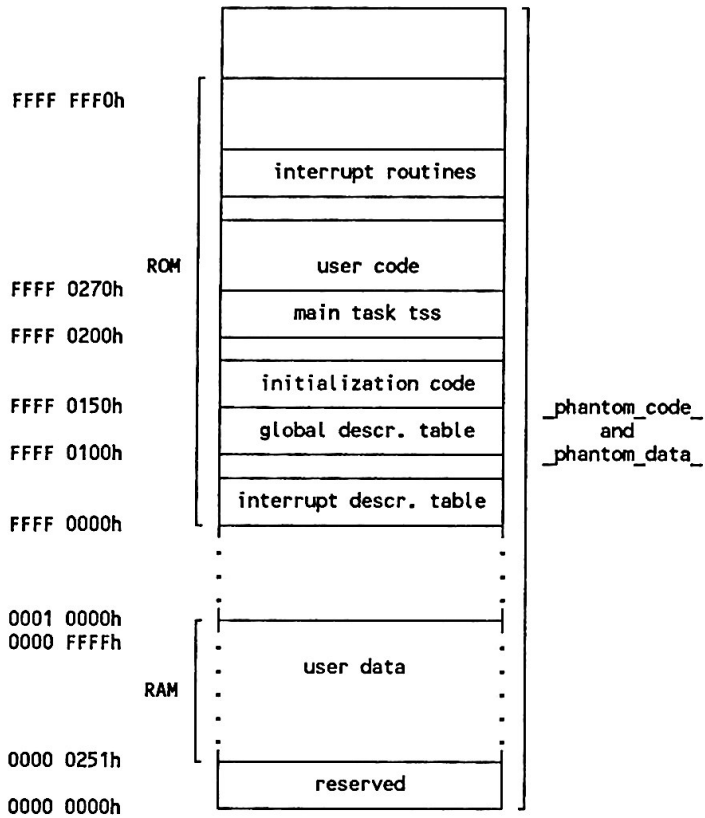


Figure 6-2 Simple Flat Model System Memory Map

6.2 Using 80386 Call Gates with C

This section discusses how to use C routines to call from one privilege level to another.

6.2.1 The Call and the Return

Calling from one privilege level to another requires the use of a call gate. To describe how to make this work with C, there must be an understanding of what goes on when calling through a call gate that goes from one privilege level to another, and returning from that call gate routine.

When a far call to the call gate is made with the intention of going from a lower privilege level to a higher privilege level, the following occurs:

1. The higher-privileged stack is checked to see if it is large enough to hold the information to be copied from the lower-privileged stack (the word count field of the call gate descriptor is checked) and to hold the stack linkage.
2. The SS and ESP from the lower-privileged stack are placed on the new higher-privileged stack.
3. The word count of the call gate determines how much information to copy from the lower-privileged stack to the new higher-privileged stack.
4. The CS and EIP are pushed onto the higher-privileged stack. The SS and ESP are now pointing to the return address on the new higher-privileged stack.
5. The code at the higher privilege level is executed.

When the far return is made, the following sequence of events occurs:

1. The CS and EIP are restored from the top of the current (higher privileged) stack, and the requested privilege level (RPL) of the CS is compared to the current privilege level (CPL).
2. If $RPL > CPL$, then an interlevel return is made. The following steps are taken for an interlevel return:
 - a. The current (higher privileged) stack's ESP is adjusted by the number of bytes indicated by the RET instruction.
 - b. The SS and ESP are set to the values from the top of the current (higher-privileged) stack.
 - c. The DS, ES, FS, and GS segment registers are checked for validity.

6.2.2 Stack Cleanup

The main effect on a C program is the execution of the RET instruction in step #2a of the return (see Section 6.2.1), where the current ESP is adjusted by the number of bytes indicated by the RET instruction. Because C does not clean up the stack, the number of bytes indicated to adjust the ESP upon return is always zero. Therefore, when returning from a C call gate routine that had parameters copied up to the higher-privileged stack, the ESP will not be adjusted to clean the parameters off the higher-privileged stack. So when step #2b of the return is performed, the SS and ESP will be set to incorrect values. Note that if no parameters are passed, there is no problem.

There are at least two ways around the stack cleanup problem:

1. Call a special interface routine that will clean the parameters off the stack before returning. This interface routine calls a C routine. With this solution, every call gate routine must have an interface procedure if the call gate routine is to be written in C.
2. When defining the call gate, give a word count of zero and use the link to the lower-privileged stack (step #2 of the call through a call gate) to get the parameters off the lower-privileged stack.

6.2.2.1 Example Interface Routines

The first source file `lowpriv.c` calls the two different call gates. The first call is to gate `C_GATE`. This step causes transfer to the procedure `ROUTINE`, which is written in C. The second source file `hipriv.c` calls the gate `INTERFACE_GATE`. This causes transfer to the interface procedure that is written in ASM386 called `INTERFACE_ROUTINE`. This procedure will then call another procedure written in C called `C_ROUTINE`.

```
/* file lowpriv.c */
/* SMALL model */

extern far c_gate();
alien far interface_gate();

main()
{
    int i;
    char ch;

    ch = 'A';
    i = 15;

    c_gate(i,ch,20);          /* Use the gate to get to ROUTINE      */

    ch = 'B';
    i = 25;

    interface_gate(i,ch,30); /* Use the gate to get to an interface */
                           /* procedure which calls C_ROUTINE */

    ch = 'C';
    i = 35;
}
```

The file `hipriv.c` contains the two call gate procedures. The first one takes advantage of the back link to the low privilege-level stack to get the passed parameters. The second procedure is called from the ASM386 interface procedure and relies on that interface procedure to take care of the stack and any parameters.

```

/* file hipriv.c */
/* SMALL model */

/* The structure "stack" is laid out like the lower-privilege level stack should */
/* look like */
/* NOTE: The variable "padding[3]" is needed because the stack is WORD- */
/* (or INT-) aligned and the structure is not. Therefore, the */
/* padding of the struct is done in the program. Since an INT is 3 */
/* bytes larger than a char, use "char padding[3]" to WORD-align */
/* the stack. */

struct stack { int param1;
               char param2;
               char padding[3];
               int param3; };

far routine(y)
    struct stack far *y;
/* y is declared to be a far pointer because */
/* the back link to the lower-privileged stack */
/* is a BASE:OFFSET combination. */
{
    int localint; /* variables to prove this works */
    char localchar;

    localint = 0;
    localint = y -> param1; /* assign parameters from the */
    localchar = y -> param2; /* lower-privileged stack and */
    localint = y -> param3; /* place into local variables */
}

/* c_routine is called from the interface procedure */

c_routine(param1, param2, param3)
    int param1;
    char param2;
    int param3;
{
    int localint; /* variables to prove this works */
    char localchar;

    localint = 0;
    localint = param1; /* do something */
    localchar = param2;
    localint = param3;
}

```

The next source file is the interface routine that will call our C procedure C_ROUTINE to do all the work. First the interface routine moves the parameters to a new part of the stack so that the C routine will have access to them. The interface routine then calls C_ROUTINE to do all the work. Most importantly, the interface routine cleans the parameters off the stack before the return.

Note that every call gate routine must have an interface procedure if the call gate routine is to be written in C.

```

/* file int_face.asm                                                    */
NAME      int_face

; module is compatible with SMALL model of segmentation

ASSUME DS:data

data      SEGMENT RW PUBLIC
dummy    DW ?                      ; avoid warning about empty segment
data      ENDS

; create a stack to use at privilege level 0

data      STACKSEG 1024H

PUBLIC interface_routine      ; make procedure accessible

code32    SEGMENT ER PUBLIC

EXTRN c_routine : NEAR

interface_routine PROC FAR      ; must be far for call gate
    PUSH EBP                    ; save EBP
    MOV  EBP,ESP
    PUSH DWORD PTR [EBP+12]     ; parameter 1
    PUSH DWORD PTR [EBP+16]     ; parameter 2
    PUSH DWORD PTR [EBP+20]     ; parameter 3
    CALL c_routine
    MOV  ESP,EBP                ; clean parameters off of stack
    POP  EBP                    ; restore EBP
    RET  0CH                    ; return and clean up parameters

interface_routine ENDP

code32    ENDS
END

```

6.2.2.2 ASM386 Startup Code

The following ASM386 source code creates the lower privilege-level stack, calls the lower-privileged routine, and provides the necessary information for the initial task state segment (TSS). Code execution begins here.

```

; file cstart.asm

NAME cstart

    ASSUME     DS:data

data    dummy    SEGMENT RW PUBLIC
data    dd ?      ; avoid warning about empty segment
        ENDS

; create the lower-privilege level stack

data    STACKSEG 1024H

code32   SEGMENT ER PUBLIC

        EXTRN  main : NEAR      ; user's code

startup:

        call main

code32   ENDS

; register initialization for initial TSS

        END startup, DS:data, SS:data

```

6.2.2.3 The Build Program

The following build program sets up the test case. First, all the segments in the module LOWPRIV are set to privilege level 3 and all the segments in HIPRIV are set to privilege level 0. Second, all the segments are placed within the global descriptor table to ensure that all routines are accessible. Third, the contents of the initial TSS are defined. Last, the names of the call gates are defined. C_GATE is a call gate and has a word count of 0 with an entry point of ROUTINE. INTERFACE_GATE is a call gate and has a word count of 3 and an entry point of INTERFACE_ROUTINE. To access these call gate procedures from a program, call the gates, not the entry points.

```

-- file callgate.bld

gate_buildfile;

SEGMENT
    lowpriv (DPL=3),
    hipriv (DPL=0);

TABLE
    GDT (ENTRY = (lowpriv,hipriv));

TASK
    init_tss (OBJECT=lowpriv,          -- Use lowpriv for start info
              STACKS=(lowpriv.data,   -- privilege level 3 stack
                      hipriv.data),   -- privilege level 0 stack
              INITIAL);

GATE
    c_gate      (CALL, DPL=3, ENTRY=routine, WC=0, USE32),
    interface_gate (CALL, DPL=3, ENTRY=interface_routine, WC=3, USE32);

END

```

6.2.2.4 The DOS Batch File

This is the DOS batch file that was used to compile, assemble, link, and build the system.

```

REM file build.bat
REM
REM ***** compile & assemble the source code *****
C386 lowpriv.c debug
C386 hipriv.c debug
ASM386 int_face.asm
ASM386 cstart.asm
REM
REM ***** BIND the low-privileged routines together *****
BND386 cstart.obj,lowpriv.obj object(lowpriv.bnd) name(lowpriv) nolo
REM
REM ***** BIND the high-privileged routines together *****
BND386 int_face.obj,hipriv.obj object(hipriv.bnd) name(hipriv) nolo
REM
REM ***** BUILD to get final result *****
BLD386 lowpriv.bnd,hipriv.bnd bf(callgate.bld) object(callgate)

```

The ICE-386 In-Circuit Emulator in "Stand-Alone/Self Test" mode was used to test this example.





Appendix A Syntactical Summary

A.1	The Build Language Syntax	A-1
A.2	The Build Program Syntax	A-2
	CREATESEG	A-2
	SEGMENT	A-2
	GATE	A-3
	TASK	A-4
	TABLE	A-5
	ALIAS	A-6
	MEMORY	A-6
	PAGING	A-7
	EXPORT	A-8

Appendix B Simple Bootloadable Files in OMF386

B.1	Overview of Bootloadable File Structure	B-1
	B.1.1 Simple OMF386 Bootloadable File Structure	B-1
	B.1.2 Bootloadable Module Header	B-2
	B.1.3 Bootloadable Partition	B-3
B.2	Bootloadable Structure with Selected Examples	B-5
	B.2.1 Bootloadable Module Header Example	B-5
	B.2.2 Bootloadable Partition with ABSTXT Example	B-7
B.3	Annotated Object File Hex Dump	B-8

Appendix C Error Messages

C.1	System-Level Exceptions	C-2
C.2	Invocation or Input Object Exceptions	C-2
C.3	Build File Messages	C-42
C.4	Internal Processing Exceptions	C-69

Appendix D Master List of Reserved Words

Appendix E Master List of Controls and Abbreviations

Appendix F VMS Software Installation

F.1	Installation Media	F-1
F.2	Execution Environment	F-1
F.3	Installation Requirements.....	F-1
F.4	Installation Procedure.....	F-2
F.5	RL386 Installation Files.....	F-5
F.6	RL386 Commands and the Help Facility	F-5
F.7	Ease of Use Kit.....	F-6

A.1 The Build Language Syntax

Characters in **bold** are literal elements of the build language. Descriptions surrounded by angle-brackets `<>` denote the described character literal. Symbols in *italics* each derive to a string of literal elements. The meta-symbol \Rightarrow is read "derives".

letter \Rightarrow { **A..Z** | **a..z** | **@** | **?** | **_** }

digit \Rightarrow { **0..9** }

hex-digit \Rightarrow { **A..F** | **a..f** | *digit* }

comment \Rightarrow -- [{ *<any character except return>* }...] *<return>*

delimiter \Rightarrow { .. | (|) | : | = | ; | , | . | - | +
| *<space>* | *<tab>* | *<return>* | *comment* }

identifier \Rightarrow *letter* [{ *letter* | *digit* }...]

dec-num \Rightarrow *digit* [{ *digit* }...]

hex-num \Rightarrow *digit* [{ *hex-digit* }...] { **H** | **h** }

number \Rightarrow { *dec-num* | *hex-num* }

number32 \Rightarrow *number*

The build language is not case-sensitive. A *number32* has a value from 0 to $2^{32}-1$. An identifier can be no more than 40 characters long. Adjacent *identifiers* and *numbers* must be separated by one or more *delimiters*.

A.2 The Build Program Syntax

Language elements which end in *-id* are *identifiers*. Elements in all caps denote literal reserved words or keywords.

$$\text{program-id} ; \left\{ \begin{array}{l} \text{createseg-definition} \\ \text{segment-definition} \\ \text{gate-definition} \\ \text{task-definition} \\ \text{table-definition} \\ \text{alias-definition} \\ \text{memory-definition} \\ \text{paging-definition} \\ \text{export-definition} \end{array} \right\} ; \dots \text{END}$$

CREATESEG

createseg-definition \Rightarrow CREATESEG *newseg* [, ...]

newseg \Rightarrow *segment-id* [(SYMBOL = *symbol-id*)]

SEGMENT

segment-definition \Rightarrow SEGMENT *seg* [, ...]

$$\text{seg} \Rightarrow \text{seg-name} \left(\left\{ \begin{array}{ll} \text{BASE} & = \text{base-addr} \\ \text{LIMIT} & = \text{segment-limit} \\ \text{ALIGN} & = \text{align-type} \\ \text{SPECLEN} & = \text{number32} \\ \text{DPL} & = \text{priv-level} \\ \text{seg-attribute} & \\ \text{use-attribute} & \end{array} \right\} \right. [, \dots] \left. \right)$$
$$\text{seg-name} \Rightarrow \left\{ \begin{array}{l} \text{module-id} \\ [\text{module-id.}] \text{combine-id} \\ \text{*SEGMENTS} \end{array} \right\}$$

base-addr \Rightarrow { *number32* | AT (*public-id*) }

segment-limit \Rightarrow [+ | -] *number32*

$$\text{align-type} \Rightarrow \left\{ \begin{array}{l} \text{BYTE} \\ \text{WORD} \\ \text{DWORD} \\ \text{QWORD} \\ \text{PARAGRAPH} \\ \text{INPAGE} \\ \text{PAGE} \end{array} \right\}$$

$$\text{priv-level} \Rightarrow \{ 0 \mid 1 \mid 2 \mid 3 \}$$

$$\text{seg-attribute} \Rightarrow \left\{ \begin{array}{l} [\text{NOT}] \text{ WRITABLE} \\ [\text{NOT}] \text{ EXPANDDOWN} \\ [\text{NOT}] \text{ READABLE} \\ [\text{NOT}] \text{ PRESENT} \\ [\text{NOT}] \text{ CONFORMING} \end{array} \right\}$$

$$\text{use-attribute} \Rightarrow \{ \text{USE16} \mid \text{USE32} \mid \text{USEREAL} \}$$

GATE

$$\text{gate-definition} \Rightarrow \text{GATE newgate } [, \dots]$$

$$\text{newgate} \Rightarrow \text{gate-id } [(\left\{ \begin{array}{l} \text{ENTRY} = \text{entry-point} \\ \text{WC} = \text{word-count} \\ \text{DPL} = \text{priv-level} \\ \text{gate-type} \\ \text{presence-indicator} \\ \text{use-attribute} \end{array} \right\} [, \dots])]$$

$$\text{entry-point} \Rightarrow \{ \text{public-id} \mid \text{task-id} \}$$

$$\text{word-count} \Rightarrow \{ 0 \mid 1 \mid \dots \mid 31 \}$$

$$\text{priv-level} \Rightarrow \{ 0 \mid 1 \mid 2 \mid 3 \}$$

$$\text{gate-type} \Rightarrow \{ \text{CALL} \mid \text{INTERRUPT} \mid \text{TRAP} \mid \text{TASK} \}$$

$$\text{presence-indicator} \Rightarrow [\text{NOT}] \text{ PRESENT}$$

$$\text{use-attribute} \Rightarrow \{ \text{USE16} \mid \text{USE32} \}$$

TASK

task-definition \Rightarrow TASK *newtask* [,...]

newtask \Rightarrow *task-id* ($\left\{ \begin{array}{l} \text{BASE} = \text{base-addr} \\ \text{CODE} = \text{code-id} \\ \text{LDT} = \text{ldt-id} \\ \text{OBJECT} = \text{module1-id} \\ \text{DPL} = \text{priv-level} \\ \text{DATA} = \text{initial-data} \\ \text{STACKS} = (\text{stack-list}) \\ \text{LIMIT} = \text{task-seg-limit} \\ \text{task-attribute} \\ \text{debug-indicator} \\ \text{presence-indicator} \\ \text{task-flags} \end{array} \right\}$ [,...])

base-addr \Rightarrow { *number32* | AT (*public-id*) }

priv-level \Rightarrow { 0 | 1 | 2 | 3 }

initial-data \Rightarrow $\left\{ \begin{array}{l} \text{module2-id} \\ [\text{module2-id.}] \text{combine1-id} \end{array} \right\}$

stack-list \Rightarrow $\left\{ \begin{array}{l} \text{module3-id} \\ [\text{module3-id.}] \text{combine2-id} \end{array} \right\}$ [,...]

task-seg-limit \Rightarrow [+] *number32*

task-attribute \Rightarrow [NOT] INITIAL

debug-indicator \Rightarrow [NOT] DEBUGTRAP

presence-indicator \Rightarrow [NOT] PRESENT

task-flags \Rightarrow $\left\{ \begin{array}{l} \text{IOPRIVILEGE} = \{ 0 | 1 | 2 | 3 \} \\ \text{[NOT] INTENABLED} \\ \text{[NOT] VIRTUALMODE} \end{array} \right\}$

TABLE

table-definition \Rightarrow TABLE *newtable* [...]

$$\text{newtable} \Rightarrow \text{table-name} \left(\left\{ \begin{array}{ll} \text{BASE} & = \text{base-addr} \\ \text{LIMIT} & = \text{table-lim} \\ \text{DPL} & = \text{priv-level} \\ \text{ENTRY} & = (\text{entry-list}) \\ \text{RESERVE} & = (\text{res-list}) \\ \text{LOCATION} & = \text{pub-id} \\ \text{presence-indicator} & \\ \text{output-indicator} & \end{array} \right\} [\dots] \right)$$

table-name \Rightarrow { GDT | IDT | *ldt-id* }

base-addr \Rightarrow { *number32* | AT (*public-id*) }

table-lim \Rightarrow [+] *slots*

slots \Rightarrow { 0 | 1 | .. | 8190 }

priv-level \Rightarrow { 0 | 1 | 2 | 3 }

$$\text{entry-list} \Rightarrow \left\{ \begin{array}{l} [\text{index} :] \text{entry} \\ \text{index} : [(\text{entry} [\dots])] \end{array} \right\} [\dots]$$

index \Rightarrow { 0 | 1 | .. | 8190 }

entry \Rightarrow { *seg-name* | *gate-id* | *task-id*[.LDT] | *anldt-id* }

$$\text{seg-name} \Rightarrow \left\{ \begin{array}{l} \text{module-id} \\ [\text{module-id.}] \text{combine-id} \end{array} \right\}$$

res-list \Rightarrow { *index1* .. *index2* } [...]

index1 \Rightarrow { 0 | 1 | .. | 8190 }

index2 \Rightarrow { 0 | 1 | .. | 8190 }

presence-indicator \Rightarrow [NOT] PRESENT

output-indicator ⇒ [NOT] CREATED

ALIAS

alias-definition ⇒ ALIAS *newalias* [...]

newalias ⇒ *main-id* (*alias-id* [...])

MEMORY

memory-definition ⇒ MEMORY *mem* [...]

mem ⇒ (*setup* [...])

setup ⇒ { [RESERVE = (*reserve-def* [...])]
[RANGE = (*range-def* [...])]
[ALLOCATE = (*alloc-def* [...])] }

reserve-def ⇒ *number1* .. *number2* [PAGED]

number1 ⇒ *number32*

number2 ⇒ *number32*

range-def ⇒ *range-id* = { RAM | ROM } (*range-list*)

range-list ⇒ { *number3* .. *number4* } [...]

number3 ⇒ *number32*

number4 ⇒ *number32*

alloc-def ⇒ *range-id* = (*name* [...])

name ⇒ { *identifier* | *TABLES | *SEGMENTS | *TASKS }

PAGING

paging-definition \Rightarrow **PAGING** *makepages*

makepages \Rightarrow *dirseg-id* ({ *maketables*
bits
LOCATION = *public-id* } [,...])

maketables \Rightarrow **PAGETABLES** *page-seg-id* [(*paging-list*)]

paging-list \Rightarrow { *range1-id*
number1 .. *number2* } [,...]

bits \Rightarrow **BITSETTING** (*range-action* [,...])

range-action \Rightarrow (*set-bit* [,...]) { *range2-id*
number3 .. *number4* }

set-bit \Rightarrow *bit-action* *bit-name*

bit-action \Rightarrow { **SET** | **RESET** | **DSET** | **DRESET** }

bit-name \Rightarrow { **P** | **RW** | **US** | **UD1** | **UD2** | **UD3** }

number1 \Rightarrow *number32*

number2 \Rightarrow *number32*

number3 \Rightarrow *number32*

number4 \Rightarrow *number32*

EXPORT

export-definition \Rightarrow EXPORT *send-out* [, ...]

send-out \Rightarrow #filename (*export-items* [, ...])

#filename \Rightarrow #< string denoting a file >

export-items \Rightarrow *export-id* (*export-list*)

export-list \Rightarrow $\left\{ \begin{array}{l} \textit{module-id} \text{ [, id-list]} \\ \textit{id-list} \text{ [, module-id] [, id-list]} \end{array} \right\}$

id-list \Rightarrow $\left\{ \begin{array}{l} \textit{gate-id} \\ \textit{public-id} \\ \textit{segment-id} \end{array} \right\} \text{ [, ...]}$

Appendix B

Simple Bootloadable Files in OMF386

A bootloadable file is used for system initialization for Intel386™ family microprocessors. Such a file contains text that can be loaded at an absolute address in memory. Bootloadable files are the default output of the Intel386 family system builder.

This appendix explains the simplest kind of bootloadable object file in Intel386 family object module format (OMF386):

- Section B.1 is an overview of the OMF386 bootloadable file structure.
- Section B.2 explains the bootloadable module header and absolute text (ABSTXT section) in greater detail, showing annotated examples from a particular bootloadable object file.
- Section B.3 contains an annotated copy of the same object file.

B.1 Overview of Bootloadable File Structure

This section is an overview with general diagrams of OMF386 bootloadable file structure. Tables B-1 through B-6 show an increasingly detailed view of its structure. The top line in each of these tables shows the number of bytes (By) in each OMF386 field. If there is an asterisk instead of a value, the field length varies from file to file.

B.1.1 Simple OMF386 Bootloadable File Structure

Bootloadable files in OMF386 begin with a module header followed by up to three sections in a single partition. The absolute text (ABSTXT) section is the only required section and the only section explained in this appendix. Table B-1 illustrates the basic structure of bootloadable OMF386 files.

Table B-1 Bootloadable Object File Structure

1 By	75 By	*	*	1 By
File Type	Module Header	Partition1	...	Checksum
B2				

The initial byte in every OMF386 file indicates the file type. The hexadecimal constant B2 specifies the OMF386 bootloadable file type.

Simple bootloadable object modules have 75 bytes of module header information, followed by one partition.

B.1.2 Bootloadable Module Header

Bootloadable module headers have seven fields, as shown in Table B-2.

Table B-2 Bootloadable Module Header

4 By	8 By	8 By	41 By	6 By	6 By	2 By
Total Space	Creat. Date	Creat. Time	Module Creator	GDT Descriptor	IDT Desc.	TSS Selector

Total Space is the minimum number of bytes needed to load the module in memory.

Creation Date is the date the module was built, written in eight characters as MM/DD/YY.

Creation Time is the approximate time the module was built, written in eight characters as HH:MM:SS (24-hour clock).

Module Creator is the version of the 80386 system builder that created the bootloadable module.

GDT Descriptor is the base address and limit for the global descriptor table.

IDT Descriptor is the base address and limit for the interrupt descriptor table.

TSS Selector is the offset of the task state segment's descriptor in the global descriptor table.

Table B-3 illustrates the format for the GDT Descriptor and IDT Descriptor fields of Table B-2.

Table B-3 GDT and IDT Fields

2 By	4 By
Limit	Linear Base Address

The GDTR (global descriptor table register) must be loaded with the GDT Descriptor field's linear base address and limit in order to load the module.

The IDTR (interrupt descriptor table register) must be loaded with the IDT Descriptor field's linear base address and limit in order to load the module.

B.1.3 Bootloadable Partition

The bootloadable module header is followed by one partition. Table B-4 illustrates OMF386 format for the partition of a simple bootloadable object module.

Table B-4 Bootloadable Partition

28 By	*	*
Table of Contents	ABSTXT Section	...

A partition consists of a table of contents followed by one or more sections. Bootloadable object modules have only one partition defined. ABSTXT is the only required section in a bootloadable

module partition. Therefore, non-required sections have been omitted in Table B-4.

The bootloadable partition's table of contents contains six location fields. The first five locations are 4-byte integers that specify the byte offset of the respective section from the start of the bootloadable object file. Table B-5 illustrates a bootloadable partition's Table of Contents field, showing its ABSTXT Location field.

Table B-5 Bootloadable Partition Table of Contents

4 By	4 By	4 By	4 By	4 By	8 By
ABSTXT Location	Reserved Location	Last Location	Next Partition	Res. Loc.	Reserved Location
	0		0	0	0

Only the ABSTXT and Last Location fields have non-zero values in simple bootloadable object files because this kind of file has only one partition with a single absolute text (ABSTXT) section. If a section is not included in the partition, then a location of 0 appears in the partition table of contents.

The ABSTXT section consists of a sequence of records. Table B-6 illustrates the format of the ABSTXT section.

Table B-6 Bootloadable Absolute Text (ABSTXT) Section

Record1:			Record2:			...
4 By	4 By	*	4 By	4 By	*	
Real Address	Length	Text	Real Addr.	Len.	Text	...

Each ABSTXT record has two fixed-length fields, followed by a text field that varies in length.

Real Address	is the absolute address where the following text field should be loaded in the 80386 address space.
Length	is the number of bytes of text that must be loaded in sequential memory to accommodate the text field.
Text	is the sequence of bytes to be loaded into memory.

B.2 Bootloadable Structure with Selected Examples

This section shows how parts of a particular bootloadable object file match the bootloadable module header and partition described in Sections B.1.2 and B.1.3. See Section B.3 for an annotated copy of the example file as a whole.

B.2.1 Bootloadable Module Header Example

Figure B-1 is a hexadecimal dump of a bootloadable object file's module header information.

```

0000  b2 08 90 01 00 31 32 2f 32 32 2f 38 37 31 32 3a
0010  33 36 3a 35 39 1a 38 30 33 38 36 20 53 59 53 54
0020  45 4d 20 42 55 49 4c 44 45 52 2c 20 58 31 31 36
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 ef 02
0040  00 01 00 00 f7 00 00 10 00 00 40 00

```

Figure B-1 Formatted Hex Dump of Bootloadable Module Header

The leftmost column is a byte count in hexadecimal, showing the offset of each line in the file. Note that the initial byte shows the B2H file type specification, indicating an OMF386 bootloadable file. The bootloadable module header begins after this byte. Each line except the first and last contains 16 bytes for a total of 75 bytes of header information. Recall the general structure illustrated by Table B-2.

Table B-2 Bootloadable Module Header

4 By	8 By	8 By	41 By	6 By	6 By	2 By
Total Space	Creat. Date	Creat. Time	Module Creator	GDT Descriptor	IDT Desc.	TSS Selector

Total Space is the 4 bytes following the file type byte, read byte by byte from right to left: 00 01 90 08H.

Creation Date is the next 8 bytes, read from left to right: 31 32 2f 32 32 2f 38 37H. These are the ASCII codes for 12/22/87.

Creation Time is the next 8 bytes, read from left to right and wrapping into the first 5 bytes of the second line: 31 32 3a 33 36 3a 35 39H. These are the ASCII codes for 12:36:59.

Module Creator is the next 41 bytes, ASCII codes for: .80386 System Builder, x116, followed by 14 zero bytes.

GDT Descriptor is the next 6 bytes, read byte by byte from right to left: 00 00 01 00 02 EFH. The GDT's linear base address is 0000100H and its limit is 02EFH (see Table B-3).

IDT Descriptor is the next 6 bytes, read byte by byte from right to left: 00 00 10 00 00 F7H. The IDT's linear base address is 00001000H and its limit is 00F7H (see Table B-3).

TSS Selector is the next 2 bytes, read byte by byte from right to left: 00 40H. This selector is an index to the descriptor beginning at the sixty-fourth byte, counting (decimal) from zero, in the GDT. Since each descriptor occupies 8 bytes, the TSS descriptor is the ninth in the global descriptor table.

B.2.2 Bootloadable Partition with ABSTXT Example

```
0040                                     68 00 00 00
0050 00 00 00 00 0d 26 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00

                                00 00 00 00 68 00 00 00
0070 00 00 00 00 c8 04 00 00 28 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00
0090 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00
00a0 00 00 00 00 00 00 00 00 c8 04 00 00 c8 04 00 00
00b0 00 00 00 00 00 00 00 00 20 00 00 00 18 00 00 00
00c0 28 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
00d0 50 00 00 00 00 00 00 00
```

Figure B-2 Formatted Hex Dump of Partition with ABSTXT Record1

The leftmost column is a byte count in hexadecimal, showing the offset of each line in the file. Note that the partition begins at the last 4 bytes of line 0040H, immediately after the module header of Figure B-1. Line 0060H has been broken in Figure B-2 to show where this partition's Table of Contents field ends and its first ABSTXT section record begins. This dump shows only the first record in the ABSTXT section. See Section B.3 for the full ABSTXT section.

Recall Tables B-5 and B-6.

Table B-5 Bootloadable Partition Table of Contents

4 By	4 By	4 By	4 By	4 By	8 By
ABSTXT Location	Reserved Location	Last Location	Next Partition	Res. Loc.	Reserved Location
	0		0	0	0

ABSTXT Location is the first 4 bytes, read byte by byte from right to left: 00 00 00 68H. This is the byte

offset of the ABSTXT section from the start of this object file.

Last Location is the third group of 4 bytes, read byte by byte from right to left: 00 00 26 0dH. This is the byte offset of the last byte in this partition.

Since only the ABSTXT Location field has a non-zero value, the Last Location field shows the byte offset of the ABSTXT section's last byte.

Table B-6 Bootloadable Absolute Text (ABSTXT) Section

Record1:			Record2:			...
4 By	4 By	*	4 By	4 By	*	
Real Address	Length	Text	Real Addr.	Len.	Text	...

Real Address is the first 4 bytes in the second half of line 0060H, read byte by byte from right to left: 00 00 00 00H. The Text field of Record1 should be loaded at absolute address 00000000H.

Length is the next 4 bytes, read byte by byte from right to left: 00 00 00 68H. The Text field of Record1 has 68H bytes.

Text is the next 68H = 104D bytes.

To load the first record in this ABSTXT section, move the 68H text bytes to absolute location 00000000H. Each subsequent record in the ABSTXT section must be loaded by moving its length in bytes of text to the absolute location specified in the record's Real Address field.

B.3 Annotated Object File Hex Dump

This section is an annotated hex dump of a simple bootloadable object module in OMF386. The leftmost column is a byte count in hexadecimal, showing the byte offset of each line in the file. An asterisk (*) in the left column indicates that one or more rows of zeros have been omitted.

File Type Byte and Module Header:

```
0000 b2 08 90 01 00 31 32 2f 32 32 2f 38 37 31 32 3a
0010 33 36 3a 35 39 1a 38 30 33 38 36 20 53 59 53 54
0020 45 4d 20 42 55 49 4c 44 45 52 2c 20 58 31 31 36
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ef 02
0040 00 01 00 00 f7 00 00 10 00 00 40 00
```

See Section B.2.1 for a detailed explanation of this part of the bootloadable file.

Partition Table of Contents:

```
0040                                     68 00 00 00
0050 00 00 00 00 0d 26 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00
```

The ABSTXT section begins at offset 68H in this file. See Section B.2.2 for a detailed explanation of this part of the bootloadable file and of the first record in the following ABSTXT section.

ABSTXT Section:

Record1 (TSS)

Real Address = 00000000H, Length = 68H = decimal 104

```
-----
0060                                     00 00 00 00 68 00 00 00
```

Text ->

```
----
0070 00 00 00 00 c8 04 00 00 28 00 00 00 00 00 00 00
0080 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 00
0090 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00
00a0 00 00 00 00 00 00 00 00 c8 04 00 00 c8 04 00 00
00b0 00 00 00 00 00 00 00 00 20 00 00 00 18 00 00 00
00c0 28 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
00d0 50 00 00 00 00 00 00 00
```

The first record is the task state segment (TSS). For this file, the TSS is loaded at absolute address 00000000H.

The second record is the global descriptor table. The record's Real Address field is 00 00 01 00H, matching the Linear Base Address field of the GDT (see Table B-3) in the module header (see Section B.2.1). The record's Length field is one byte more than the GDT Limit field. The TSS selector in the module header specifies the offset of the TSS descriptor in this GDT.

Record2 (GDT)

Real Address = 0000100H, Length = 2F0H = decimal 752

00d0 00 01 00 00 f0 02 00 00

Text ->

00e0 00 00 00 00 00 00 00 00 ef 02 00 01 00 92 00 00
00f0 f7 00 00 10 00 92 00 00 a3 00 f0 03 00 9a 40 00
0100 56 00 00 00 01 92 40 00 c7 04 58 00 01 92 40 00
0110 ff 0f 00 20 00 92 40 00 ff 0f 00 30 00 92 40 00

TSS descriptor

0120 67 00 00 00 00 89 00 00

more GDT descriptors

 ff 9f 00 01 00 92 41 00
0130 17 00 68 00 00 82 00 00 00 00 00 00 00 00 00
0140 00 00 00 00 00 00 00 00 06 00 00 90 01 92 40 00
0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

*

The third record is the interrupt descriptor table. Its Real Address field (1000H) matches the module header's (see Section B.2.1) IDT Linear Base Address field (see Table B-3), and its Length field is one byte more than the IDT Limit field.

Record3 (IDT)

Real Address = 00001000H, Length = F8H = decimal 248

03d0 00 10 00 00 f8 00 00 00

Text ->

00 00 00 00 00 00 00 00
03e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0440 00 00 00 00 00 00 00 00 24 00 18 00 00 8e 00 00
0450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*

The fourth record is the local descriptor table, and the remaining records are code/data.

Record4 (LDT)

Real Address = 00000068H, Length = 18H = decimal 24

04d0 68 00 00 00 18 00 00 00

Text ->

00 00 00 00 00 00 00 00
04e0 17 00 68 00 00 92 00 00 e5 00 94 04 00 f2 40 00

Record5 (code/data)

Real Address = 00003000H, Length = 1000H = decimal 4096

04f0 00 30 00 00 00 10 00 00

Text ->

```

                                07 06 00 00 07 12 00 00
0500  07 22 00 00 07 32 00 00 07 42 00 00 07 52 00 00
0510  07 62 00 00 07 72 00 00 07 82 00 00 07 92 00 00
0520  07 a2 00 00 07 b2 00 00 07 c2 00 00 07 d2 00 00
0530  07 e2 00 00 07 f2 00 00 07 00 01 00 07 10 01 00
0540  07 20 01 00 07 30 01 00 07 40 01 00 07 50 01 00
0550  07 60 01 00 07 70 01 00 07 80 01 00 06 00 00 00
0560  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

*

14f0 00 00 00 00 00 00 00 00

Record6 (code/data)

Real Address = 00002000H, Length = 1000H = decimal 4096

00 20 00 00 00 10 00 00

Text ->

```

1500  07 30 00 00 00 00 00 00 00 00 00 00 00 00 00
1510  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

*

Record7 (code/data)

Real Address = 00010004H, Length = 50H = decimal 80

2500 04 00 01 00 50 00 00 00

Text ->

```

                                00 20 00 00 0d 0a 59 6f
2510  75 20 68 61 76 65 20 6a 75 73 74 20 63 61 75 73
2520  65 64 20 61 20 70 61 67 65 20 66 61 75 6c 74 2c
2530  0d 0a 00 0d 0a 50 61 67 65 20 6c 6f 61 64 65 64
2540  20 66 6f 72 20 79 6f 75 2c 20 63 6f 6e 74 69 6e
2550  75 69 6e 67 2e 2e 2e 00

```

Record8 (code/data)

Real Address = 000003f0H, Length = A1H = decimal 161

f0 03 00 00 a1 00 00 00

Text ->

2560 a1 04 00 00 00 0f 22 d8 0f 20 c0 0d 01 00 00 80
2570 0f 22 c0 b8 68 00 00 00 8e e8 65 8b 1d 00 00 00
2580 00 cc 00 00 5a 66 60 b9 26 00 00 00 bb 08 00 00
2590 00 66 ff 33 9a 12 00 00 00 28 01 43 e2 f3 b8 48
25a0 00 00 00 8e e0 0f 20 d0 c1 e8 16 c1 e0 02 0f 20
25b0 d9 01 c8 64 8b 18 81 e3 00 f0 ff ff 0f 20 d0 c1
25c0 e8 7b c1 e0 02 25 ff 0f 00 00 01 c3 64 8b 0b 81
25d0 e1 ff 0f 00 00 0f 20 d0 25 00 f0 3f 00 09 c8 0d
25e0 01 00 00 00 64 89 03 b9 24 00 00 00 bb 2f 00 00
25f0 00 66 ff 33 9a 12 00 00 00 28 01 43 e2 f3 66 61
2600 cf

Record9 (code/data)

Real Address = 00019000H, Length = 4H

00 90 01 00 04 00 00 00

Text ->

ae 6f 38 a0

Checksum

dc

The Checksum is the last byte of every OMF386 file.



Appendix C

Error Messages

BLD386 issues a message when it encounters one of the following conditions:

- **WARNING:** although a possible programming error exists, the output object file is valid.
- **ERROR:** the output object file is probably invalid even though BLD386 processing can continue.
- **FATAL ERROR:** BLD386 processing aborts. All open files are closed. The object file created, if any, is not completed.

Digits that accompany messages indicate the location of the exception.

- **No digit:** exception is at the system interface level.
- **100-199, 400-499:** exception is in the invocation line or in an input object file.
- **200-299, 500-599:** exception is in the build file.
- **300-399:** exception is in internal BLD386 processing.

Messages appear in the print file and any errorprint file. Fatal error messages also appear at the console.

This appendix provides up to four kinds of information for each message:

- **MEANING:** how to interpret the message
- **CAUSE:** the probable reason for the message
- **EFFECT:** the state of the output file(s) and the status of BLD386
- **ACTION:** suggestions for correcting the condition

Messages are listed in numeric order as grouped above.

C.1 System-Level Exceptions

SYSTEM INTERFACE ERROR

error text

FILE: *file name*

MEANING: A fatal error occurred in a call to the host operating system. The *error text* contains a message issued by the operating system. The *file name* is present if the error is an I/O error.

CAUSE: Problems such as an I/O error, invalid parameters, or insufficient memory can cause this condition.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Refer to host operating system documentation for interpretation. Correct the error and restart BLD386.

C.2 Invocation or Input Object Exceptions

ERROR 100: INPUT FILE MISSING

MEANING: This fatal error occurred because no linkable input file was provided.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386 with at least one linkable input file.

ERROR 101: PATHNAME TOO LONG

NEAR: *token string*

MEANING: This fatal error occurred because too many characters are in a file name in the invocation line near the *token string*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386 using a valid file name.

ERROR 102: MISSING LEFT PARENTHESIS

NEAR: *token string*

MEANING: This fatal error occurred because a left parenthesis is missing after the *token string*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Insert a left parenthesis in the proper location when you reinvoke BLD386.

ERROR 103: MISSING RIGHT PARENTHESIS

NEAR: *token string*

MEANING: This fatal error occurred because a right parenthesis is missing after the *token string*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Insert a right parenthesis in the proper location when you reinvoke BLD386.

ERROR 104: DUPLICATE FILE NAME IN INPUT LIST

FILE: *file name*

MEANING: This fatal error occurred because *file name* exists in more than one place in the invocation line's input list.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386, ensuring that none of the input file names matches.

ERROR 105: FILE ALREADY SPECIFIED IN COMMAND TAIL

FILE: *file name*

MEANING: This fatal error occurred because the *file name* exists in more than one place in the invocation line. At least one of the duplicate file names is explicit in or implied by the controls.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386, ensuring that none of the file names, explicit or default, matches.

ERROR 106: INVALID DELIMITER IN COMMAND TAIL
NEAR: *token string*

MEANING: This fatal error occurred because the invocation line contained an improperly placed delimiter or used an illegal character as a delimiter. The invalid delimiter was detected either before or after the *token string*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386 using valid delimiters in the invocation line. Valid delimiters include left and right parentheses and commas.

ERROR 107: INVALID TOKEN IN COMMAND TAIL
NEAR: *token string*

MEANING: This fatal error occurred because the invocation line contained a *token string* that BLD386 could not interpret.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386 using a correct *token string*.

ERROR 108: TOKEN TOO LONG
NEAR: *token string*

MEANING: This fatal error occurred because the *token string* contained too many characters, e.g., a module name exceeding 40 characters.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386 using a token string with a valid length.

ERROR 109: UNKNOWN CONTROL IN COMMAND TAIL

NEAR: *token string*

MEANING: This fatal error occurred because one of the controls in the invocation line is invalid. The invalid control is contained in *token string*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386, correctly spelling the control or its abbreviation.

ERROR 110: DUPLICATE MODULE NAME

NEAR: *token string*

MEANING: This fatal error occurred because the module name is specified more than once in the invocation line.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386, ensuring that a module name is not specified in the invocation line more than once.

ERROR 111: LINE TOO LONG IN CONTROL FILE

FILE: *file name*

MEANING: This fatal error occurred because the control file specified by *file name* contains a line longer than 128 characters.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Correct the control file, ensuring that all lines, excluding comments, are less than 128 characters long.

ERROR 112: NESTED CONTROL FILES

FILE: *file name*

MEANING: This fatal error occurred because the control file identified by *file name* contains a reference to another control file.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Eliminate the nesting in the control file; then reinvoke BLD386.

WARNING 113: NOT ENOUGH MEMORY FOR SORTING

MEANING: This warning occurred because BLD386 does not have sufficient user memory (RAM) to sort gate and/or task names.

EFFECT: BLD386 processing continues. The gates and/or tasks listed in the print file's gate table and/or task table, respectively, are not in alphabetical order.

ERROR 114: NUMBER OF SYMBOLS EXCEEDS INTERNAL LIMIT

MEANING: This fatal error occurred because the maximum number of symbols BLD386 can process has been exceeded.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Try invoking BLD386 with fewer input modules or use BND386 to purge public symbols from input modules; then reinvoke BLD386.

ERROR 115: WORKFILE ERROR DURING SYMBOL PROCESSING

FILE: *workfile file name*

MEANING: This fatal error indicates that BLD386 cannot create a temporary file to use.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Ensure all temporary files created by the operating system are deleted; then reinvoke BLD386.

WARNING 116: ILLEGAL PRIVILEGE OF INITIAL DS OR SS

TASK: *task name*

MEANING: This warning indicates that the privilege level of the initial data or stack segment is not equal to that of the initial code segment.

EFFECT: BLD386 fills the appropriate TSS fields with the values specified.

ACTION: Revise the input modules as necessary.

ERROR 117: INVALID OBJECT FILE

FILE: *file name*

MODULE: *module name*

MEANING: This fatal error indicates that the module specified by *module name*, contained in the file referred to by *file name*, has an invalid format.

CAUSE: This condition may have been caused by an internal translator or utility error. This condition may also have been caused by using loadable or bootloadable modules as input.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Revise the input list or retranslate the source files with 80386 translators to create linkable input modules for BLD386, relinking with BND386 if necessary. If the problem persists, contact your Intel representative.

ERROR 118: INPUT SEGMENTS EXCEED TARGET MEMORY

Not used.

WARNING 119: OVERLAP BETWEEN SEGMENTS OR TABLES

LOW ADDRESS: *32-bit number*

HIGH ADDRESS: *32-bit number*

MEANING: This warning indicates that items being written to the output bootloadable module have overlapping addresses.

CAUSE: Conflicts exist in the build file between base addresses, segment limits, and/or reserved memory.

EFFECT: BLD386 continues processing, but the object file could be invalid.

ACTION: Pinpoint the location of the overlap with the aid of the segment map in the print file. Redefine bases, limits, and ranges, as necessary.

ERROR 120: INVALID TABLE LIMIT

TABLE: *table name*

MEANING: This error condition indicates that a table limit specified in a TABLE definition in the build file is either too small to accommodate all the table entries specified or too large to be accommodated in one segment.

EFFECT: Although BLD386 processing continues, the output object module is invalid.

ACTION: Reinvoke BLD386 after correcting the TABLE definition in the build file.

ERROR 121: MISMATCHED ATTRIBUTES FOR DATA AND STACK

FILE: *file name*

MODULE: *module name*

SEGMENT: *segment name*

MEANING: This error condition exists because BLD386 detected data and stack segments with the same *segment name*, but with incompatible access rights. The data and stack segments are both contained in the module identified by *module name*, housed in *file name*.

CAUSE: Although BLD386 does not function as a linker, it does attempt to combine data and stack segments with the same name when they are in the same module (e.g., PL/M-286 SMALL model). The resulting segment is called a DSC (data/stack combined) segment.

EFFECT: Although BLD386 processing continues, the output object module may not be usable.

ACTION: Assign data and stack segments compatible attributes, or retranslate segments to place them in different modules. Reinvoke BLD386 with the adjusted input file(s).

ERROR 122: SAME SEGMENT PLACED AT TWO DIFFERENT ENTRIES
IN LDT

FILE: *file name*

MODULE: *module name*

SEGMENT: *segment name*

MEANING: This error condition exists because a descriptor for the segment identified by *segment name*, contained in *module name* and *file name*, has been installed more than once in one LDT.

CAUSE: The input list may contain multiple export modules with ambiguities.

EFFECT: Although BLD386 processing continues, the output object module is not usable.

ACTION: Correct any export module(s) used in the invocation; then reinvoke BLD386.

ERROR 123: ENTRY POINT IN EXPORTED GATE NOT ACCESSIBLE
VIA GDT

GATE: *gate name*

MEANING: This error occurred because the gate specified in an EXPORT definition does not point to a segment that has a descriptor installed in the GDT.

EFFECT: BLD386 processing continues, but the export module cannot be used.

ACTION: Redefine the build file EXPORT definition or other parameters to ensure that the export module is valid.

ERROR 124: EXPORTED SYMBOL NOT ACCESSIBLE VIA GDT

SYMBOL: *symbol name*

MEANING: This error occurred because the public symbol named in an EXPORT definition does not occur in a segment that has a descriptor installed in the GDT.

EFFECT: BLD386 processing continues, but the export module cannot be used.

ACTION: Redefine the build file EXPORT definition or other parameters to ensure that the export module is valid.

ERROR 125: SAME SYMBOL DEFINED TO BE IN DIFFERENT SEGMENTS

FILE: *file name*

MODULE: *module name*

SEGMENT: *segment name*

SYMBOL: *symbol name*

MEANING: This error condition occurred because *symbol name* is defined in a segment (identified by *segment name*) different from the segment name used in the public definition. The module and file that contain the symbol name declaration are defined by *module name* and *file name*, respectively.

EFFECT: Although BLD386 processing continues, the output object module is invalid.

ACTION: Correct the source file public or external declarations; then reinvoke BLD386 after retranslating and possibly relinking with BND386.

WARNING 126: SYMBOL TYPES MISMATCH

FILE: *file name*

MODULE: *module name*

SYMBOL: *symbol name*

MEANING: This warning condition exists because the *symbol name* in *module name* and *file name* has the same name as a symbol in an input module, but is not of the same type.

EFFECT: BLD386 processing continues, but the output module is not usable.

ACTION: Correct the type of mismatched symbol or change one of the *symbol name*; then reinvoke BLD386.

WARNING 128: SPECIFIED MODULE NOT FOUND IN INPUT FILE
FILE: *file name*
MODULE: *module name*

MEANING: This warning indicates that *module name* was specified in the input list, but could not be found in the associated file, identified by *file name*.

EFFECT: BLD386 processing continues.

ACTION: If the referenced module must be included in the output module, reinvoke BLD386 using a file containing the module.

ERROR 129: CS REGISTER INITIALIZED BY NONEXECUTABLE
SEGMENT
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This error condition indicates that *segment name* is not executable, but is identified in the main module named by *module name* as the initial code segment. The module, typically an ASM386 module, is named by *module name* and resides in *file name*.

EFFECT: BLD386 processing continues, but the output object module is not usable.

ACTION: Ensure that CS register initialization requirements are satisfied in the input module(s); then reinvoke BLD386.

ERROR 130: SS REGISTER INITIALIZED BY NONWRITABLE
SEGMENT

FILE: *file name*

MODULE: *module name*

SEGMENT: *segment name*

MEANING: This error condition indicates that *segment name* is not writable, but is identified in the main module named by *module name* as an initial stack segment. The module, typically an ASM386 module, is named by *module name* and resides in *file name*.

EFFECT: BLD386 processing continues, but the output module is unusable.

ACTION: Ensure that SS register initialization requirements are satisfied in the input module(s); then reinvoke BLD386.

ERROR 131: REFERENCE TO UNRESOLVED EXTERNAL SYMBOL

FILE: *file name*

MODULE: *module name*

REFERRING LOCATION: *location*

REFERENCED LOCATION: *target*

MEANING: This error condition indicates that input modules do not contain a public definition identified by *target*, which is referred to as an external symbol in the segment named by *location*. The external declaration occurs in *module name* of *file name*.

EFFECT: Although BLD386 processing continues, the output object module may not be usable.

ACTION: Reinvoke BLD386 using a file that will resolve the external reference.

ERROR 132: REFERENCED LOCATION BEYOND LIMIT
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error exists because the entire *target* information is not in the referenced segment, probably because a segment limit value is too small. The referring *location* is housed in *module name* and *file name*.

EFFECT: BLD386 processing continues, but the output module is not valid.

ACTION: Correct the build file to ensure access to referenced information. Reinvoke BLD386.

ERROR 133: REFERRING AND REFERENCED LOCATIONS BELONG TO DIFFERENT LDTs
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error condition exists because a descriptor for the segment containing *target* is installed in an LDT different from that housing a descriptor referring to the *target*. The segment containing the reference is identified by *location* and is housed in *module name* and *file name*.

EFFECT: BLD386 processing continues, but the output object module is not usable.

ACTION: Adjust the build file specifications to make the target location accessible to the referring location, e.g., via a gate or a GDT entry. Then reinvoke BLD386 using the revised build file.

ERROR 134: REFERENCED LOCATION IS AT INACCESSIBLE
LOCATION

FILE: *file name*

MODULE: *module name*

REFERRING LOCATION: *location*

REFERENCED LOCATION: *target*

MEANING: This error condition exists because the target is housed in a code segment that has a DPL numerically greater than that of the segment named in *location*. The segment containing the reference is identified by *location* and is housed in *module name* and *file name*.

EFFECT: Although BLD386 processing continues, the output object module is not usable.

ACTION: Revise the build file to ensure that the access is in accord with protected-mode privilege rules; then reinvoke BLD386 using the revised build file.

ERROR 135: ENTRY POINT SPECIFIED IN GATE IS
NONEXECUTABLE

FILE: *file name*

MODULE: *module name*

REFERRING LOCATION: *location*

REFERENCED LOCATION: *target*

MEANING: This error condition exists because the entry point identified by *target* is housed in a nonexecutable segment. Gates are used to mediate access to code segments only.

EFFECT: BLD386 processing continues, but the output object module is not usable.

ACTION: Revise the build file specifications to make the access possible, or revise any export module(s) used in the invocation; then reinvoke BLD386.

ERROR 136: ENTRY POINT SPECIFIED IN GATE IS LESS
PRIVILEGED THAN GATE
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error condition exists because the DPL of the *target* location is numerically greater than that of a gate referred to in *location*. A gate can only point to a code segment at a DPL equal to or numerically less than that of the gate itself.

EFFECT: BLD386 processing continues, but the output object module is not valid.

ACTION: Revise the build file to reflect privilege rules or revise any export module(s) used in the invocation; then reinvoke BLD386.

ERROR 137: BAD SELF RELATIVE REFERENCE
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error condition occurred because *target* and *location* are not in the same segment. The *location* is housed in *module name* in *file name*.

CAUSE: This condition may have been caused by a translator error or by using erroneous ASM386 code.

EFFECT: BLD386 processing continues, but the output object module cannot be used.

ACTION: Retranslate the module; then reinvoke BLD386 with the new input file.

ERROR 138: REFERRING LOCATION BEYOND LIMIT
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error exists because the *location* referring to *target* is outside the limits of any segment. The *location* is housed in *module name* in *file name*.

EFFECT: BLD386 processing continues, but the output module is not valid.

ACTION: Correct the build file to ensure access to referenced information, adjusting segment limits if necessary; then reinvoke BLD386.

ERROR 139: IMPROPER ENTRY POINT SPECIFICATION FOR GATE
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error condition occurred because the entry point in the gate named in *location* is neither a GDT selector nor a reference to a segment.

EFFECT: BLD386 processing continues, but the output object module cannot be used.

ACTION: Revise the build file to reflect privilege rules or change any export module(s) used in the invocation. Then reinvoke BLD386.

ERROR 140: NUMBER OF AUTOMATIC GATES EXCEEDS INTERNAL
LIMIT

FILE: *file name*

MODULE: *module name*

REFERRING LOCATION: *location*

REFERENCED LOCATION: *target*

MEANING: This fatal error condition occurred because BLD386 automatically created over 8190 gates. The reference to the last gate created is defined by *location*, *module name*, and *file name*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reexamine the scheme used to make interlevel references, making build file specifications for gates where appropriate; then reinvoke BLD386 using the new build file.

ERROR 141: AUTOMATIC GATE INTRODUCED

FILE: *file name*

MODULE: *module name*

REFERRING LOCATION: *location*

REFERENCED LOCATION: *target*

MEANING: This warning indicates that BLD386 automatically created a call gate to mediate an access between *location* and *target*. The location is housed in *module name* of *file name*.

EFFECT: BLD386 processing continues, and the output module is valid.

ACTION: Ensure that the created gate affects the desired access.

ERROR 142: REFERENCE TO AN ENTRY IN IDT IS NOT ALLOWED
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error occurred because BLD386 detected a reference in *location* to a gate or segment that points to *target*, which is installed in an IDT. The *location* is part of the module named by *module name* in a file identified by *file name*.

EFFECT: BLD386 processing continues, but the output module cannot be used.

ACTION: Revise the input module and/or the build file to ensure that no explicit references to trap or interrupt gates exist; these gates can only be invoked as the result of an external interrupt. Reinvoke BLD386 with the new input and/or build file.

WARNING 143: NO SPECIFICATION FOR GDT

MEANING: This warning indicates that TABLE definition specifications for a GDT are not included in the input.

EFFECT: BLD386 processing continues, and the output module is valid.

ACTION: Ensure that GDT specifications are not required for this invocation.

WARNING 144: NO SPECIFICATION FOR IDT

MEANING: This warning indicates that TABLE definition specifications for an IDT are not included in the input.

EFFECT: BLD386 processing continues, and the output module is valid.

ACTION: Ensure that IDT specifications are not required for this invocation.

ERROR 145: TEXT FOUND IN STACK SEGMENT
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This error indicates that the stack segment named by *segment name* in *module name* contains text.

EFFECT: BLD386 processing continues, but the object module could be invalid.

ACTION: Ensure the result is acceptable; reinvoked BLD386 if necessary.

WARNING 146: NO TASK SPECIFIED IN INPUT

MEANING: This warning indicates that no TASK definition specification exists in the build file, and no main module occurs in the input.

EFFECT: BLD386 processing continues, and the output object module is valid.

ACTION: Correct the build file or the input file(s); then reinvoked BLD386.

ERROR 147: REFERENCE TO SYMBOL ON STACK FOUND
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This warning indicates that the *target* referenced in *location* resides in a stack segment. The referring *location* is contained in a module called *module name* in a file called *file name*.

EFFECT: BLD386 processing continues, and the effect on the output object module is not known.

ACTION: Correct the input module if necessary; then reinvoked BLD386.

ERROR 148: REFERENCE TO INTERRUPT OR TRAP GATE FOUND
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error occurred because BLD386 detected a reference in *location* to a trap or interrupt gate. The *location* is part of the module named by *module name* in a file identified by *file name*.

EFFECT: BLD386 processing continues, but the output module cannot be used.

ACTION: Revise the input module and/or the build file to ensure that no explicit references to trap or interrupt gates exist; these gates can be invoked only as the result of an external interrupt. Reinvoke BLD386 with the new input and/or build file.

ERROR 149: CONSTANT VALUE OVERFLOW
FILE: *file name*
MODULE: *module name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error occurred because the sum of a constant value represented by a public symbol at *target* and an incremental value in the instruction at *location* caused a byte or word overflow.

EFFECT: BLD386 processing continues, but the output may not be usable.

ACTION: Reassign values, if required; then reinvoke BLD386.

WARNING 150: EXPORTED SEGMENT NOT ACCESSIBLE VIA GDT
SEGMENT: *segment name*

MEANING: This warning indicates that an export module created contains a segment, identified by *segment name*, which has a descriptor that is not installed in the GDT of the loadable or bootloadable module created in the same invocation.

EFFECT: BLD386 processing continues, and the export module is valid.

ACTION: Ensure that the desired result has been achieved; reinvoke BLD386, if necessary, to redefine the export module.

ERROR 151: TEXT LENGTH IS GREATER THAN SEGMENT LENGTH
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This fatal error indicates that the limit of the segment named by *segment name* is not large enough. The segment is contained in *module name* in *file name*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Revise the build file to correct the segment limit; then reinvoke BLD386.

ERROR 152: CS REGISTER NOT INITIALIZED
FILE: *file name*
MODULE: *module name*

MEANING: This error condition exists because BLD386 could not find an initialization value for the CS register in the main module named by *module name* in *file name*. This error usually occurs with input that includes ASM286 or ASM386 main modules.

EFFECT: BLD386 processing continues, but the output object module is not valid.

ACTION: Provide CS initialization information; then reinvoke BLD386.

ERROR 153: SS REGISTER NOT INITIALIZED

FILE: *file name*

MODULE: *module name*

MEANING: This error condition exists because the main module named by *module name* does not contain SS register initialization information. This error usually occurs with input that includes ASM286 or ASM386 main modules.

EFFECT: BLD386 processing continues, but the output module is not valid.

ACTION: Provide SS initialization information; then reinvoke BLD386.

ERROR 154: DS REGISTER NOT INITIALIZED

FILE: *file name*

MODULE: *module name*

MEANING: This error condition exists because the main module identified by *module name* does not contain DS initialization information. This error usually occurs with input that includes ASM286 or ASM386 main modules.

EFFECT: BLD386 processing continues even though the output object module is not usable.

ACTION: Provide DS initialization information; then reinvoke BLD386.

ERROR 155: SPECIFIED TABLE LOCATION BEYOND SEGMENT LIMIT

TABLE: *table name*

MEANING: This error condition exists because the LOCATION construct in the TABLE definition for *table name* names a public symbol that does not represent a 6-byte location.

EFFECT: BLD386 processing continues, but the output module is not valid.

ACTION: Ensure that a 6-byte location is made available, by redefining the limit of the segment that houses the public symbol; then reinvoke BLD386.

ERROR 156: UNRESOLVED EXTERNAL SYMBOL IN REGISTER
INITIALIZATION
FILE: *file name*
MODULE: *module name*
SYMBOL: *symbol name*

MEANING: This error occurred because *symbol name*, which defines an initialization value, has no public definition in the input.

EFFECT: BLD386 processing continues, but the output is not valid.

ACTION: Provide initialization information; then reinvoke BLD386.

ERROR 157: REFERENCE TO AN AMBIGUOUS SEGMENT
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This error occurred because the input modules contain at least three segments with the name identified by *segment name*, and one of them is an empty segment. One of the segments is housed in *module name* and *file name*.

EFFECT: BLD386 processing continues, but the output cannot be used.

ACTION: Process with BND386 the modules that contain duplicate segment names or rename the segments; then reinvoke BLD386.

ERROR 158: REFERENCE TO AN EMPTY SEGMENT
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This error occurred because a reference to an empty segment identified by *segment name* exists in the input.

EFFECT: BLD386 processing continues, but the output may not be valid.

ACTION: Ensure that the reference is correct; reinvoke BLD386 with different input modules if necessary.

ERROR 159: CANNOT FIND SPECIFIED MODULE TO BE EXPORTED
FILE: *file name*
MODULE: *module name*

MEANING: This error exists because an EXPORT definition names a module identified by *module name* that is not present in the input. The *file name* is the name of the file that was to contain the export module.

EFFECT: BLD386 processing continues, but the export module may not be usable.

ACTION: Redefine the EXPORT definition or modify the input list; then reinvoke BLD386.

ERROR 160: ONLY ONE INPUT MODULE MAY BE EXPORTED TO ONE EXPORT FILE
FILE: *file name*
MODULE: *module name*

MEANING: This error exists because an EXPORT definition contains more than one module identifier. The export module is housed in file name, and *module name* identifies the second module identifier in the definition.

EFFECT: BLD386 processing continues, but the export module may not be usable.

ACTION: Redefine the EXPORT definition; then reinvoke BLD386.

ERROR 161: CONFLICTING GATE EXPORT
FILE: *file name*
MODULE: *module name*
SYMBOL: *symbol name*

MEANING: This fatal error occurred because an EXPORT definition contains a gate identifier and a public identifier with the same symbol name. The *module name* is the module identifier in the EXPORT definition, and the *file name* identifies the file that was to contain the export module.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Revise the EXPORT definition to eliminate the name conflict; then reinvoke BLD386.

WARNING 162: DUPLICATE PUBLIC SYMBOLS CANNOT BE EXPORTED
FILE: *file name*
MODULE: *module name*
SYMBOL: *symbol name*

MEANING: This warning indicates that an EXPORT definition contains more than one public identifier with the name identified by *symbol name*. The export module is identified by *module name*. The file that contains the export module is identified by *file name*.

EFFECT: BLD386 processing continues, and the export module created contains only one occurrence of the public symbol.

ACTION: If necessary, revise the EXPORT definition and reinvoke BLD386.

ERROR 163: EXPORTING MULTIPLE MODULES TO ONE FILE NOT SUPPORTED
FILE: *file name*
MODULE: *module name*
SYMBOL: *symbol name*

MEANING: This error occurred because an EXPORT definition contains more than one export module. The *module name* identifies the second export module in the definition. The *file name* is the name of the export file.

EFFECT: BLD386 processing continues, and all export module specifications except the first one in the definition are ignored.

ACTION: If necessary, reinvoke BLD386 after modifying the EXPORT definition. Ensure that each definition contains only one export module specification.

ERROR 164: INPUT HAS TWO MODULES WITH SAME NAME
FILE: *file name*
MODULE: *module name*

MEANING: This error occurred because two input files contain modules with the same *module name*. The second occurrence of the *module name* was detected in *file name*.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386, ensuring that the input modules contain no duplicate names.

ERROR 165: AMBIGUOUS EXTERNAL-PUBLIC RESOLUTION

FILE: *file name*

MODULE: *module name*

SEGMENT: *segment name*

SYMBOL: *symbol name*

MEANING: This error occurred because a public definition was found in a segment with the same name as at least two other segments in the input, and one of them was an empty segment. The public symbol is identified by *symbol name*, and the segment by *segment name*. The *module name* and *file name* house the segment that contains the public definition.

EFFECT: BLD386 processing continues, but the output may not be usable.

ACTION: If necessary, use BND386 to eliminate the duplicate segment name condition or rename the segments; then reinvoked BLD386.

ERROR 166: AMBIGUOUS REGISTER INITIALIZATION

FILE: *file name*

MODULE: *module name*

SEGMENT: *segment name*

MEANING: This error occurred because register initialization information was found in a segment with the same name as at least two other segments in the input, and one of them was an empty segment. The *segment name* identifies the name of the ambiguous segment. The *module name* and *file name* house the segment that contains the initialization information.

EFFECT: BLD386 processing continues, but the output may not be usable.

ACTION: If necessary, use the BND386 to eliminate the duplicate segment name condition, or rename the segments; then reinvoked BLD386.

ERROR 167: REGISTER INITIALIZED BY AN EMPTY SEGMENT
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This error occurred because the segment identified by *segment name* is expected to contain register initialization information, but is empty.

EFFECT: BLD386 processing continues, but the output is not valid. One of the following messages also appears: 152, 153, or 154.

ACTION: Provide adequate initialization information in the input segments; then reinvoke BLD386.

ERROR 168: REFERENCED GATE IS AT HIGHER PRIVILEGE
FILE: *file name*
MODULE: *module name*
GATE: *gate name*
REFERRING LOCATION: *location*
REFERENCED LOCATION: *target*

MEANING: This error condition exists because the DPL of the gate identified by *gate name* is numerically lower than that of the segment that contains the *location*. The entry point specified in the gate is identified by *target*.

EFFECT: BLD386 processing continues, but the output is not usable.

ACTION: Revise the conditions under which the CALL occurs in order to follow the rules of privilege; then reinvoke BLD386.

ERROR 169: UNDEFINED RESULTS DUE TO TWO INPUT MODULES WITH SAME NAME
FILE: *file name*
MODULE: *module name*

MEANING: This fatal error occurred because the input contains two modules with the same name, creating ambiguity in internal BLD386 processing.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reinvoke BLD386 after eliminating the duplicate name situation.

ERROR 170: SEGMENT OVERFLOW IN DATA AND STACK
COMBINATION
FILE: *file name*
MODULE: *module name*
SEGMENT: *segment name*

MEANING: This error occurred because a combination of data and stack segments identified by *segment name* would result in a segment larger than 64K bytes. The segments are housed in *module name* and *file name*.

EFFECT: BLD386 processing continues, but the output is not valid.

ACTION: Modify the size of the input segments or rearrange their contents; then reinvoked BLD386.

ERROR 171: OUT OF TABLE SLOTS FOR AUTOMATIC GATE

MEANING: This fatal error can occur for two reasons:

- 1) a descriptor table limit has been reached.
- 2) REAL segments are present which disallow creation of automatic gates.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Revise the build file to avoid the condition. If REAL segments are present, add GATE definitions to replace those that BLD386 tried to create automatically.

ERROR 172: SYMBOL TYPE INFORMATION IS TOO COMPLEX
FILE: *file name*

MEANING: In processing symbol types, BLD386 has internal limits on the size of an individual type definition it can process. If the limit is exceeded, BLD386 truncates the type definition to a null type and issues this error.

EFFECT: BLD386 processing continues and truncates the size as specified. The output object file is valid.

ACTION: Identify the symbol definition(s) in the source program that might have caused this error message. Change the symbol declaration so that a smaller type definition is generated (refer to the appropriate compiler manual), then recompile the necessary source program before reinvoking BLD386.

ERROR 173: SEGMENT FOR THE ENTRY IN GATE NOT EXPORTED
FILE: *file name*
MODULE: *module name*
GATE: *gate name*

MEANING: In EXPORT definition, *gate name* is exported. Although the gate has been exported, the segment identifier that represents the entry in the gate is not exported.

EFFECT: BLD386 processing continues. The exported module may not be usable, and the output object file is valid.

ACTION: Rewrite the GATE definition portions of the build file, being sure that the segment described is also exported; then reinvoke BLD386.

ERROR 174: GATE AND ITS ENTRY POINT ARE IN DIFFERENT LDTs
GATE: *gate name*

MEANING: In GATE definition, *gate name* is a gate that has been placed in one LDT and the entry point refers to a segment that has been placed in a different LDT.

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Rewrite the GATE definition portions of the build file, being sure each gate and its entry are always in the same LDT or GDT; then reinvoke BLD386.

WARNING 175: TABLE ALLOCATED OUTSIDE SPECIFIED RANGE
TABLE: *table name*

MEANING: In MEMORY definition, the RANGE and ALLOCATE constructs were both used, but a descriptor table was not allocated in any range. Therefore, BLD386 attempts to allocate the descriptor table in a range that had the attribute ROM. The warning is issued because BLD386 cannot find enough space in any ROM range, or because no ROM range is specified by the user.

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Rewrite the MEMORY definition portions of the build file, if necessary. Then reinvoke BLD386.

WARNING 176: TASK ALLOCATED OUTSIDE SPECIFIED RANGE
TASK: *task name*

MEANING: In MEMORY definition, the RANGE and ALLOCATE constructs were both used, but the task named by *task name* was not allocated in any range. Therefore, BLD386 attempts to allocate the task in a range that had the attribute ROM. The warning is issued because BLD386 cannot find enough space in any ROM range, or because you did not specify a ROM range.

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Rewrite the MEMORY definition portions of the build file; then reinvoke BLD386.

WARNING 177: SEGMENT ALLOCATED OUTSIDE SPECIFIED RANGE
SEGMENT: *segment name*

MEANING: In MEMORY definition, the RANGE and ALLOCATE constructs were both used, but the segment named by *segment name* was not allocated in any range. Therefore, BLD386 attempts to allocate the segment in a range that had the attribute corresponding to the access right of the segment. The warning is issued because BLD386 cannot find enough space in any such range, or because you did not specify a range with the required attributes.

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Rewrite the MEMORY definition portions of the build file; then reinvoke BLD386.

WARNING 178: SEGMENT ACCESS CONFLICTS WITH MEMORY TYPE
SEGMENT: *segment name*

MEANING: In MEMORY definition, the RANGE and ALLOCATE constructs were both used, but the access rights of the segment named by *segment name* conflicted with those of the range in which the segment is allocated (e.g., a read-write segment allocated in a ROM range).

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Rewrite the MEMORY definition portions of the build file; then reinvoke BLD386.

ERROR 179: CONFLICTING ADDRESS FOR ALIASING SEGMENT
SEGMENT: *segment name*

MEANING: In MEMORY definition, the specified segment was allocated within a range and was also included as an alias in an ALIAS definition. This represents a conflict in addressing schemes.

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Examine the MEMORY definition and ALIAS definition lines and correct them, as required; then reinvoke BLD386.

ERROR 180: BOOTSTRAP SYMBOL DOES NOT EXIST
NAME: *symbol name*

MEANING: The *symbol name* specified with the BOOTSTRAP control does not exist as a public symbol in the input module.

EFFECT: BLD386 processing continues. The output object file is valid.

ACTION: Reinvoke BLD386 using a *symbol name* that exists as a public symbol in the input module(s).

ERROR 181: BOOTSTRAP SYMBOL DOES NOT BELONG TO A SEGMENT
SYMBOL: *symbol name*

MEANING: The *symbol name* specified with the BOOTSTRAP control is not a public symbol that resides in a segment. For example, the symbol could be a public constant value or a gate.

EFFECT: BLD386 processing continues. The output object file is valid. The jump instruction is not placed at location 0FFFFFFF0H.

ACTION: Reinvoke BLD386 using a *symbol name* that exists as a public symbol and resides in a segment.

WARNING 182: BOOTSTRAP SYMBOL NOT IN LAST 64KB OF MEMORY
SYMBOL: *symbol name*

MEANING: The *symbol name* specified with the BOOTSTRAP control has an address such that it cannot be accessed from location 0FFFFFFF0H with a near jump.

EFFECT: BLD386 processing continues. BLD386 still creates the jump instruction, but its effect is doubtful. The output object file is valid.

ACTION: Reinvoke BLD386 using a *symbol name* such that the symbol's address lies in the range 0FFFF0000H to 0FFFFFFFCH.

ERROR 183: BOOTSTRAP SYMBOL IS NOT PUBLIC
NAME: *symbol name*

MEANING: The *symbol name* specified with the BOOTSTRAP control is an external symbol, not resolved by any other public symbol in the input.

EFFECT: BLD386 processing continues, ignoring the BOOTSTRAP control. The output object file is valid.

ACTION: Reinvoke BLD386 using a *symbol name* that exists as a public symbol in the input module(s).

WARNING 184: BOOTSTRAP ADDRESS NOT IN LAST 64KB OF MEMORY

ADDRESS: *number*

MEANING: The absolute address *number* specified with the BOOTSTRAP control cannot be accessed from location 0FFFFFFF0H with a near jump.

EFFECT: BLD386 processing continues. BLD386 still creates the jump instruction and places it at location 0FFFFFFF0H, but its effect is doubtful. The output object file is valid.

ACTION: Reinvoke BLD386 using an address that lies within the range 0FFFF0000H to 0FFFFFFFCH.

WARNING 185: BOOTSTRAP SYMBOL DOES NOT BELONG TO AN EXECUTABLE SEGMENT

SYMBOL: *symbol name*

MEANING: The *symbol name* specified with the BOOTSTRAP control does not reside in an executable segment. An attempt to jump to such a symbol generates a hardware fault.

EFFECT: BLD386 processing continues. BLD386 still creates the jump instruction and places it at location 0FFFFFFF0H, but its effect is doubtful. The output object file is valid.

ACTION: Reinvoke BLD386 using a *symbol name* that resides in an executable segment.

ERROR 186: COMBINING DIFFERENT ATTRIBUTES

MEANING: Two segments, data and stack, are being combined to form a DSC segment; each segment has a different 80286/80386 attribute.

EFFECT: The attribute is set to USE16 by BLD386.

ACTION: Correct USE16/USE32 attributes of the combining segments; then reinvoke BLD386.

ERROR 187: INPAGE ALIGNED SEGMENT TOO LONG, TRUNCATED TO
1 PAGE

MEANING: A segment whose alignment parameter is defined as INPAGE must fit within a page frame, therefore it cannot be more than one page, or 4K bytes, long.

EFFECT: This requirement is normally intended for data segments e.g., TSSs that must be paragraph aligned and should not be split on a page swap. INPAGE is used to prevent a segment that crosses a page boundary from being split into two pages as the result of a page swap.

ACTION: Recheck requirements of alignment and limit parameters; then reinvoke BLD386.

ERROR 188: RELDESC NOT ALLOWED WITH BOOTLOAD - IGNORED

MEANING: The invocation line control RELDESC is only valid for dynamically loadable output.

EFFECT: BLD386 processing continues and the control RELDESC is ignored. The output object file is valid.

ACTION: Correct the invocation line; then reinvoke BLD386.

ERROR 189: CANNOT CREATE LODFIX RECORD - IGNORED

MEANING: This error should never occur.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Contact your Intel representative. An error has occurred that is internal to your system or software that cannot be corrected by a user.

ERROR 190: DEBUG SEGMENT CANNOT ALSO HAVE BSS

MEANING: A segment that has BSS area also has combine code DEBUG.

EFFECT: None.

ACTION: This error should never occur, check output from the language translator that produced this input file; then reinvoke BLD386.

ERROR 191: NO SPACE FOR TABLE - BASE NOT SET
TABLE: *name of descriptor table*

MEANING: This error indicates that insufficient memory was found to contain the table.

CAUSE: There is not enough memory to assign an address to the table.

EFFECT: No base address could be calculated for the table. The output object file is invalid.

ACTION: Reinvoke BLD386 with the corrected input or build file.

ERROR 192: NO SPACE FOR SEGMENT - BASE NOT SET
SEGMENT: *name of segment*

MEANING: This error indicates that insufficient memory was found to contain the segment.

CAUSE: There is not enough memory to assign an address to the segment.

EFFECT: No base address could be calculated for the segment. The output file is invalid.

ACTION: Reinvoke BLD386 with the corrected input or build file.

ERROR 193: NO SPACE FOR TASK - BASE NOT SET

TASK: *name of task*

MEANING: This error indicates insufficient memory was found to contain this task.

CAUSE: There is not enough memory to assign an address to the task.

EFFECT: No base address could be calculated for the task. The output file is invalid.

ACTION: Reinvoke BLD386 with the corrected input or build file.

ERROR 194: WRONG ALIGNMENT FOR SETTING DIRECTORY BIT

LOW ADDRESS: *32-bit number*

HIGH ADDRESS: *32-bit number*

MEANING: This error indicates an attempt was made to set a bit in the page directory for a range that is not a multiple of 4M bytes.

EFFECT: BLD386 processing continues. The request is not honored. The output object file is valid.

ACTION: Redefine the bitsetting request correctly; then reinvoke BLD386.

ERROR 195: RANGE FOR BITSETTING OVERLAPS KNOWN RANGES
LOW ADDRESS: 32-bit number
HIGH ADDRESS: 32-bit number

MEANING: This error indicates a memory range in the BITSETTING construct of the PAGING definition was defined that does not coincide with the limits of paged memory. This error can also be generated when setting the UD1 bit of segments that have a non-zero SPECLLEN in a SEGMENT definition.

EFFECT: BLD386 processing continues. The BITSETTING request is not honored. The output object file is valid.

ACTION: Check the definition of paged memory and the addresses reported. Use the build file listing and segment map to pinpoint the offending BITSETTING request or segment. Note that this error can also be caused by fragmentation of the internal BLD386 database by successive BITSETTING requests. For example, the following encompasses two disjoint blocks and would cause such an error:

```
BITSETTING ( (SET UD1) 1000h..2ffffh,  
             (SET UD2) 3000h..4ffffh,  
             (SET UD3) 1000h..7ffffh )
```

Order the bitsetting requests so that no one request bridges disjoint blocks as follows:

```
BITSETTING ( (SET UD3) 1000h..7ffffh,  
             (SET UD2) 3000h..4ffffh,  
             (SET UD1) 1000h..2ffffh )
```

ERROR 196: VIRTUAL/REAL MIX WRONG FOR TASK OR GATE HAS REAL ENTRY

FILE: *file name*
MODULE: *module name*
TABLE: *table name*

MEANING: Either the entry point for a gate is in a real segment or a virtual TSS has initial segments that are not real or a non-virtual TSS has initial segments that are real.

EFFECT: Contents of gate or TSS unpredictable.

ACTION: Correct build file and rerun BLD386.

WARNING 197: MIXED REAL/NOT REAL FIXUP

MEANING: The referring and referred locations of a fixup are not both in real segments or not real segments.

EFFECT: The fixup of the referred location will be according to its type, that is, the selector if the type is USE16/USE32 or the paragraph number if the type is USEREAL.

ACTION: Check build file, correct and resubmit if necessary.

ERROR 198: FLAT OPTION WITH NOBOOTLOAD OPTION OR USE16

MEANING: A USE16 segment is in the input object module, or USE16 is specified in a build file definition, or the NOBOOTLOAD control is used with the FLAT control. USE16 segments are not supported in FLAT model. FLAT model is supported only in the bootloadable output of BLD386. FLAT model is not supported in the loadable output of BLD386.

EFFECT: BLD386 aborts processing.

ACTION: Do not specify NOBOOTLOAD and FLAT in the same invocation. If you use the FLAT control, ensure that a USE16 segment is not in an input object file, and that USE16 is not specified in a build file definition.

ERROR 199: 80376 DOES NOT SUPPORT USE16 CODE SEGMENTS

MEANING: The 80376 does not support USE16 code segments in the input object file.

EFFECT: BLD386 processing continues but the object file will be invalid.

ACTION: Correct the source; then reinvoke BLD386.

WARNING 400: SEGMENTS WILL OVERLAP ON THE 80376

MEANING: Controls on the command line or from an input object file will create a segment larger than the maximum address. Because the 80376 has an address bus smaller than 32 bits, addresses greater than the bus size will wrap to low memory. This could overlay segments that are in low memory. See the MOD376 control in Chapter 4.

EFFECT: BLD386 continues processing, but the object file could be invalid.

ACTION: Correct source if needed; then reinvoke BLD386.

WARNING 401: FEWER THAN 32 INTERRUPTS ARE DEFINED

MEANING: The first 32 interrupts are Intel-reserved. However, if a limit value is specified in the TABLE definition for the IDT, then the builder will use that limit value even if it specifies fewer than 32 entries. If no limit value is specified, or if there is no TABLE definition for the IDT, then the IDT will default to 32 entries. If the limit specifies more than 32 IDT entries, then the builder uses the specified number. The object file will be valid.

EFFECT: BLD386 continues processing. The object file is valid.

ACTION: Correct the TABLE definition if needed; then reinvoke BLD386.

WARNING 402: OVERLAPPING TEXT

SYMBOL: *symbol name*

LOW ADDRESS: *32-bit number*

HIGH ADDRESS: *32-bit number*

MEANING: Items being written to the output bootloadable module have text which is overlapping. This means that when the module is loaded, text which originated from *symbol name* will overwrite text already loaded at physical addresses between LOW ADDRESS and HIGH ADDRESS. Note that the 80376 has an address bus smaller than 32 bits. Addresses that do not overlap on the 80386 might overlap on the 80376. Therefore if the MOD376 control is specified, then the reduced bus size of the 80376 may be the reason this warning is issued.

EFFECT: The specified text will be loaded over text already loaded from LOW ADDRESS to HIGH ADDRESS. The object file may be invalid.

ACTION: Inspect the map file to determine the sources of overlapping text and correct the build file or correct the input object modules; then reinvoke BLD386.

ERROR 403: TABLE ALLOCATED OUTSIDE PHANTOM DATA SEGMENT
TABLE: *table name*

MEANING: The specified table is not within the limit of the data phantom segment.

EFFECT: Build processing continues but the object file is invalid.

ACTION: Extend the limit value of the phantom data segment in the build file, or reduce the limit of other segments in the build file, or reduce the size of input segments, or some combination thereof. Then reinvoke BLD386.

ERROR 404: TASK ALLOCATED OUTSIDE PHANTOM DATA SEGMENT
TASK: *task name*

MEANING: The specified task is not within the limit of the phantom data segment.

EFFECT: Build processing continues but the object file is invalid.

ACTION: Extend the limit value of the phantom data segment in the build file, or reduce the limit of other segments in the build file, or reduce the size of input segments, or some combination thereof. Then reinvoke BLD386.

ERROR 405: SEGMENT ALLOCATED OUTSIDE PHANTOM DATA
SEGMENT
SEGMENT: *segment name*

MEANING: The specified segment is not within the limit of the phantom data segment.

EFFECT: Build processing continues but the object file is invalid.

ACTION: Extend the limit value of the phantom data segment in the build file, or reduce the limit of other segments in the build file, or reduce the size of input segments, or some combination thereof. Then reinvoke BLD386.

ERROR 406: SEGMENT ALLOCATED OUTSIDE PHANTOM CODE
SEGMENT

SEGMENT: *segment name*

MEANING: The specified segment is not within the limit of the phantom code segment.

EFFECT: Build processing continues but the object file is invalid.

ACTION: Extend the limit value of the phantom code segment in the build file, or reduce the limit of other segments in the build file, or reduce the size of input segments, or some combination thereof. Then reinvoke BLD386.

ERROR 407: EXPAND DOWN STACK DESCRIPTOR CREATED WITH THE FLAT OPTION

MEANING: By default, in flat model all stack segments are not expanddown and based relative to `_phantom_data_`. You can specify an expanddown stack segment that is not based relative to `_phantom_data_`. This message indicates that a separate stack was created.

EFFECT: BLD386 processing continues. The object file is valid.

ACTION: No action is required if this features was desired. If you do not want a separate stack, refer to the information on separate stacks in the FLAT control in Chapter 4 and correct the build file.

C.3 Build File Messages

When these messages appear in the build file listing section of the print file, they have the following format:

```
* * * WARNING 2xx: LINE n, NEAR token, message
```

Where:

2xx is the number associated with the message.

n is the number of the build file listing line on which the error occurred.

token is a string pointing to the location of the error.

message contains the text of the message.

The numbers and text associated with build file messages are defined as follows:

```
WARNING 200: ILLEGAL TOKEN(S) SKIPPED UNTIL token
```

MEANING: This warning indicates that the build file contains an invalid character string.

EFFECT: BLD386 ignores the line up to *token*.

ACTION: Correct the build file, and reinvoke BLD386.

WARNING 201: MISSING (INSERTED
WARNING 202: MISSING) INSERTED
WARNING 203: MISSING : INSERTED
WARNING 204: MISSING - INSERTED
WARNING 205: MISSING ; INSERTED
WARNING 206: MISSING , INSERTED
WARNING 207: MISSING . INSERTED
WARNING 208: MISSING .. INSERTED
WARNING 209: MISSING - INSERTED
WARNING 210: MISSING + INSERTED

MEANING: These warnings are provided because the delimiter identified was missing but required for valid syntax.

EFFECT: BLD386 inserts the required delimiter in the necessary location.

ACTION: Ensure that BLD386 corrections complement your objectives.

WARNING 211: MISSING NUMBER
WARNING 212: MISSING IDENTIFIER
WARNING 213: MISSING ATTRIBUTE
WARNING 214: MISSING GATE TYPE

MEANING: These warnings indicate that required information was omitted from a specification line.

EFFECT: BLD386 ignores the part of the specification that depends on the missing information.

ACTION: Revise the build file to provide the missing information; then reinvoke BLD386.

WARNING 215: MISSING PARAMETER KEYWORD INSERTED

MEANING: This warning indicates that a keyword is missing.

EFFECT: BLD386 inserts a keyword.

ACTION: Ensure that the BLD386 correction is appropriate.

WARNING 216: TOO LONG IDENTIFIER TRUNCATED

MEANING: This warning indicates that an identifier used had more than 40 characters.

EFFECT: BLD386 truncates the identifier to 40 characters.

ACTION: Ensure that the truncation does not cause undesirable results.

WARNING 217: TOO LONG TOKEN TRUNCATED

MEANING: This warning indicates that the length of a token exceeded 45 characters.

EFFECT: BLD386 truncates the token to 45 characters.

ACTION: Ensure that the truncation does not cause undesirable results.

WARNING 218: PARAMETER ALREADY SPECIFIED

MEANING: This warning indicates that a specification is provided more than once in a data structure definition.

EFFECT: BLD386 uses the first specification encountered.

ACTION: Ensure that the desired result was achieved.

WARNING 219: ILLEGAL VALUE FOR BASE

MEANING: This warning indicates that the segment base specified is invalid.

EFFECT: BLD386 ignores the specification.

ACTION: Correct the base address; then reinvoke BLD386.

WARNING 220: ILLEGAL VALUE FOR LIMIT

MEANING: This warning indicates that the segment limit is less than the original size of the segment, or that the limit value reduces the size of a descriptor table or a TSS, or that the segment limit is greater than 65,535.

EFFECT: BLD386 ignores the SEGMENT definition. The output object file is valid.

ACTION: Correct the segment limit in the SEGMENT definition line(s) of the build file; then reinvoke BLD386.

WARNING 221: ILLEGAL VALUE FOR PRIVILEGE LEVEL

MEANING: This warning indicates that the privilege level specified is invalid or greater than 3.

EFFECT: BLD386 assumes a privilege of level 3.

ACTION: Ensure that the default privilege level is appropriate.

WARNING 222: ILLEGAL VALUE FOR WORD COUNT

MEANING: This warning occurred because the word count specified in GATE definition for a call gate is invalid or greater than 31.

EFFECT: BLD386 extracts the word count from the input segment used in the ENTRY construct.

ACTION: Ensure that the default word count value is appropriate.

WARNING 223: ENTRY NOT FOUND

MEANING: This warning occurred because an ENTRY construct in a TABLE definition could not be located in the input module(s) or among preceding data structure definitions.

EFFECT: BLD386 ignores the specification.

ACTION: Correct the invocation conditions to identify the entity to be installed prior to installing the entity.

WARNING 224: MODULE NOT FOUND IN INPUT

MEANING: This warning indicates that the input list did not contain a module referenced in a module identifier.

EFFECT: BLD386 ignores the definition that contains the specification.

ACTION: Revise the build file and/or the input list to resolve this condition.

WARNING 225: SEGMENT NOT FOUND IN INPUT

MEANING: This warning indicates that the input list did not contain a segment referenced in a combine identifier.

EFFECT: BLD386 ignores the definition that contains the specification.

ACTION: Revise the build file and/or the input list to resolve this condition.

WARNING 226: ILLEGAL DESCRIPTOR ATTRIBUTE

MEANING: This warning occurred because the descriptor type specification was not compatible with other attributes of the same descriptor.

EFFECT: BLD386 ignores the specification that contains this error.

ACTION: Revise the input module and/or the build file to eliminate this condition.

WARNING 227: SEGMENT ALREADY EXISTS

MEANING: This warning indicates that the table or TSS being defined was defined in an earlier specification.

EFFECT: BLD386 ignores the specification.

ACTION: Correct the build file if necessary; then reinvoke BLD386.

WARNING 228: PUBLIC SYMBOL NOT FOUND IN INPUT

MEANING: This warning occurred because the public identifier could not be located among the input segments.

EFFECT: The definition using the public identifier is ignored.

ACTION: Correct the build file and/or the input list; then reinvoke BLD386.

WARNING 229: REGISTER INITIALIZATION NOT FOUND IN INPUT

MEANING: This warning indicates that the module identified by a module identifier in the OBJECT construct of a TASK definition does not contain the required register initialization information for the TSS.

EFFECT: BLD386 fills the following TSS fields with zeroes: CS selector, EIP (entry point), DS selector, and possibly SS selector and ESP.

ACTION: If necessary, retranslate the input module and/or reinvoke BLD386 after providing sufficient initialization information.

WARNING 230: TASK NOT FOUND

MEANING: This warning indicates that a task identifier used in a GATE definition or a TABLE definition was not defined in a TASK definition.

EFFECT: BLD386 ignores the definition.

ACTION: Revise the build file to ensure that the TSS is defined before it is referenced in a later specification.

WARNING 231: LDT NOT FOUND

MEANING: This warning occurred because an LDT identifier used in a TASK definition does not correspond to a previously defined LDT.

EFFECT: BLD386 ignores the definition.

ACTION: Revise the build file to ensure that the LDT is defined (via a TABLE definition) before it is referenced in a later specification.

WARNING 232: UNGATABLE PUBLIC

MEANING: This warning indicates that a GATE definition for a call gate uses a public identifier with an illegal word count or a word count greater than 31, or that the public identifier is not in a code segment.

EFFECT: BLD386 ignores the GATE definition.

ACTION: Ensure that the correct public identifier is referenced; then reinvoke BLD386.

WARNING 233: GATE ALREADY EXISTS

MEANING: This warning occurred because the GATE definition uses a gate identifier assigned in an earlier GATE definition.

EFFECT: BLD386 implements only the first specification.

ACTION: Recreate the GATE definition if necessary; then reinvoke BLD386.

WARNING 234: TASK ALREADY EXISTS

MEANING: This warning means that the TASK definition uses a task identifier assigned in an earlier TASK definition.

EFFECT: BLD386 implements only the first specification.

ACTION: Recreate the TASK definition if necessary; then reinvoke BLD386.

WARNING 235: STACK ALREADY SPECIFIED

MEANING: This warning indicates that more than one STACKS construct exists for a stack at a given privilege level.

EFFECT: BLD386 implements only the first specification.

ACTION: Ensure that the stacks at all four levels are initialized properly; modify the build file if necessary.

WARNING 236: ILLEGAL VALUE FOR RANGE

MEANING: This warning indicates that an index value specified in the reserve list of a TABLE definition is invalid or greater than the maximum allowed (8190 for GDT and LDT, 255 for IDT).

EFFECT: BLD386 ignores the reserve list specification.

ACTION: Correct the index value in the build file; then reinvoke BLD386.

WARNING 237: ILLEGAL INDEX VALUE

MEANING: This warning means that an index value specified in the entry list of a TABLE definition is invalid or greater than the maximum allowed (8190 for GDT and LDT, 255 for IDT).

EFFECT: BLD386 ignores the index specification.

ACTION: Correct the index value in the build file; then reinvoke BLD386.

WARNING 238: TOO MANY ENTRIES

MEANING: This fatal error indicates that the table limit has been exceeded.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Reevaluate the build file specifications that define the table; make changes and reinvoke BLD386 as necessary.

WARNING 239: TABLE ALREADY EXISTS

MEANING: This warning occurred because the TABLE definition includes a table name or index already defined.

EFFECT: BLD386 ignores this specification.

ACTION: Revise the build file if necessary; then reinvoke BLD386.

WARNING 240: ILLEGAL TABLE LOCATION

MEANING: This warning indicates that the public identifier specified in the **LOCATION** construct of a **TABLE** definition is not located in a readable segment, or that the table is absolute.

EFFECT: BLD386 ignores the **LOCATION** construct.

ACTION: Revise the build file if necessary; then reinvoke BLD386.

WARNING 241: ILLEGAL PRIVILEGE OF INITIAL STACK SEGMENT

MEANING: This warning means that the privilege level of the initial stack segment is not equal to that of the initial code segment.

EFFECT: BLD386 fills the appropriate TSS fields with the specified values.

ACTION: Revise the build file and/or input modules if necessary.

WARNING 242: ILLEGAL PRIVILEGE OF INITIAL DATA SEGMENT

MEANING: The privilege level of the initial data segment is numerically less than that of the initial code segment.

EFFECT: BLD386 fills the appropriate TSS fields with the specified values.

ACTION: Revise the build file and/or input modules if necessary.

WARNING 243: UNDEFINED INITIAL CODE SEGMENT

MEANING: This warning indicates that an **INITIAL** construct in a **TASK** definition identified a public identifier that did not contain initial CS and EIP register values for the task.

EFFECT: BLD386 fills the appropriate TSS fields with zeros.

ACTION: Revise the build file and/or input modules if necessary.

WARNING 244: EXPORT ITEM NOT FOUND

MEANING: This warning indicates that one of the items to be exported in accord with an **EXPORT** definition could not be located in the input or among entities defined earlier in the invocation.

EFFECT: BLD386 processing continues, and the output is valid.

ACTION: Ensure that any item referenced in an **EXPORT** definition is accessible to BLD386 when it processes that definition.

WARNING 245: FILE ALREADY EXISTS

MEANING: This warning indicates that a file with the file name specified in an **EXPORT** definition already exists.

EFFECT: BLD386 processing continues, and the **EXPORT** definition is ignored.

ACTION: Respecify the file name in the **EXPORT** definition; then reinvoke BLD386.

WARNING 246: ILLEGAL VALUE FOR MEMORY RANGE

MEANING: This warning indicates that the **RANGE** construct in a **MEMORY** definition is invalid.

EFFECT: BLD386 processing continues, and BLD386 ignores the **MEMORY** definition.

ACTION: Redefine the definition, ensuring that the numbers specified in the **RANGE** construct are valid 32-bit values, and that the proper punctuation is used. Reinvoke BLD386.

WARNING 247: SEGMENT IS EMPTY

MEANING: This warning indicates that a build language specification refers to an empty segment.

EFFECT: BLD386 processing continues, and the output is valid.

ACTION: Ensure that the reference to the empty segment is acceptable; revise the input and reinvoke BLD386 if required.

WARNING 248: SEGMENT IS AMBIGUOUS

MEANING: This warning indicates that the segment identifier refers to at least three input segments, one of which is an empty segment.

EFFECT: BLD386 processing continues, but the output module may not be valid.

ACTION: Rename input segments, process them using BND386, or use a module identifier to precisely identify a specific segment in a given module; then reinvoke BLD386.

WARNING 249: SYMBOL IS NOT PUBLIC

MEANING: The specification in the definition requires a public symbol to be named.

EFFECT: BLD386 processing continues, and the specification is ignored.

ACTION: Revise the definition as required; then reinvoke BLD386.

WARNING 250: ILLEGAL IDT ENTRY

MEANING: A TABLE definition contains a specification for installing a descriptor other than an interrupt, trap, or task gate.

EFFECT: BLD386 processing continues, and the specification is ignored.

ACTION: Respecify the entry list, ensuring that only interrupt, trap, or task gates are installed in the IDT. Reinvoke BLD386.

WARNING 251: ILLEGAL IDT OR LDT ENTRY

MEANING: A TABLE definition contains a specification for installing an LDT or a TSS descriptor.

EFFECT: BLD386 processing continues, and the specification is ignored.

ACTION: Respecify the entry list, ensuring that only valid descriptors are installed in the LDT or IDT. Only task, interrupt, or trap gates can be installed in an IDT. Only segment descriptors, call gates, and task gates can be installed in an LDT.

WARNING 252: ILLEGAL GDT OR LDT ENTRY

MEANING: A TABLE definition contains a specification for installing a trap or interrupt gate.

EFFECT: BLD386 processing continues, and the specification is ignored.

ACTION: Respecify the entry lists, ensuring that only valid descriptors are installed in the GDT or LDT. Neither trap nor interrupt gates can be installed in these tables.

WARNING 253: TABLE INDEX CONFLICT

MEANING: This warning indicates that an entry list specification in a TABLE definition conflicts with a reserve list specification in the same or another TABLE definition.

EFFECT: BLD386 processing continues. The last specification for the slot is implemented.

ACTION: Ensure that the desired effect has been achieved; redefine the TABLE definition and reinvoke BLD386 if required.

WARNING 254: SEGMENT IS NOT WRITABLE

MEANING: This warning indicates that an operation valid only for writable segments is specified for a nonwritable segment; for example, the SS register is initialized with a nonwritable segment.

EFFECT: BLD386 processing continues, but the output module is not valid.

ACTION: Modify the build file specifications to ensure that the SS register is initialized with a writable segment; then reinvoke BLD386.

WARNING 255: SEGMENT IS NOT EXECUTABLE

MEANING: This warning indicates that the CS register is initialized with a nonexecutable segment.

EFFECT: BLD386 processing continues, but the output module is not valid.

ACTION: Modify the build file specifications to ensure that the CS register is initialized with an executable segment.

WARNING 256: MAIN DESCRIPTOR NOT FOUND

MEANING: The main descriptor name specified in an ALIAS definition does not exist in the input or was not declared in the build file.

EFFECT: BLD386 processing continues. The aliasing that was requested does not occur. The output object file is valid.

ACTION: Specify a main descriptor name in BLD386's input or in the build file; then reinvoke BLD386.

WARNING 257: MAIN DESCRIPTOR IS ITSELF AN ALIAS

MEANING: The main descriptor name specified in an ALIAS definition has been specified as an alias segment name in a previous ALIAS definition for some other entity—table, task or segment. This usage is treated as a "nested" alias. Nested aliasing—the aliasing of an alias—is not allowed by BLD386.

EFFECT: BLD386 processing continues. The aliasing that was requested does not occur. The output object file is valid.

ACTION: Remove the offending ALIAS definition. Use another approach to solve the memory-management problem for which this ALIAS definition was intended; then reinvoke BLD386.

WARNING 258: CANNOT ALIAS BY MODULE NAME

MEANING: The alias segment name in an ALIAS definition is also the name of an input module. BLD386 does not support the concept of using all the segments in a module to alias an entity.

EFFECT: BLD386 processing continues. The aliasing that was requested does not occur. The output object file is valid.

ACTION: Remove the offending ALIAS definition. Use another aliasing method; then reinvoke BLD386.

WARNING 259: SPECIFIED ALIAS NOT WRITABLE

MEANING: The segment specified by an alias segment name in an ALIAS definition is not writable. Allowing aliasing for a segment that is not writable is inconsistent with the intent of BLD386's definition of aliasing.

EFFECT: BLD386 processing continues. The aliasing that was requested does occur as intended. The output object file is valid.

ACTION: Make sure that aliasing is as intended. If necessary, make appropriate changes; then reinvoke BLD386.

WARNING 260: PUBLIC SYMBOL IS NOT IN A SEGMENT

MEANING: The symbol in the BASE construct of a SEGMENT, TABLE, or TASK definition does not reside in a segment. The public symbol could be a public constant value or a gate.

EFFECT: BLD386 processing continues. The segment base definition does not occur as intended. The output object file is valid.

ACTION: Rewrite the SEGMENT definition line or lines. Ensure that the public identifier belongs to a segment that has already been defined; then reinvoke BLD386.

WARNING 261: PUBLIC SYMBOL NOT IN ABSOLUTE SEGMENT

MEANING: The symbol in the BASE construct of a SEGMENT definition does not reside in an absolute segment.

EFFECT: BLD386 processing continues. The segment base definition does not occur as intended. The output object file is valid.

ACTION: Rewrite the SEGMENT definition line or lines. Ensure that the public identifier belongs to a segment that has already been assigned its absolute address; then reinvoke BLD386.

WARNING 262: POSSIBLE STACK PRIVILEGE CONFLICT

MEANING: The code specified by the CODE construct in TASK definition is a conforming code segment. The purpose of the warning is to alert you to the special situations that may occur if the initial code is conforming.

EFFECT: BLD386 processing continues. The desired TASK definition occurs as intended. The output object file is valid.

ACTION: Because the code is conforming, the task may be executed at any privilege level. Ensure that a stack is initialized for each privilege level as appropriate.

WARNING 263: RANGE NOT FOUND

MEANING: In MEMORY definition, the ALLOCATE construct refers to a range name that has not been previously defined. This warning is not issued if BLD386 detects that no items in the input module remain to be allocated in that range.

EFFECT: BLD386 processing continues. The desired use of the ALLOCATE construct does not take place. The output object file is valid.

ACTION: Rewrite the MEMORY definition. Ensure that the RESERVE, RANGE, and ALLOCATE constructs are used in the indicated order; then reinvoke BLD386.

WARNING 264: DESCRIPTOR DOES NOT EXIST

MEANING: In MEMORY definition, the ALLOCATE construct refers to an item identified by a name. However, the item was neither specified in the build file nor does it exist in the input modules.

EFFECT: BLD386 processing continues. The desired use of the ALLOCATE construct does not take place. The output object file is valid.

ACTION: Rewrite the MEMORY definition. Be sure that the item referred to in the ALLOCATE construct (a segment, table, or task) is either specified in the build file or exists in the input module(s); then reinvoke BLD386.

WARNING 265: ACCESS CONFLICTS WITH MEMORY TYPE

MEANING: In MEMORY definition, the ALLOCATE construct refers to an item and a range. However, the item and range have mismatched attributes. For example, this warning would be issued if a segment with read-write access was allocated in a memory range with a ROM (read-only) attribute, as specified in the RANGE construct.

EFFECT: BLD386 processing continues. The desired use of the ALLOCATE construct takes place. The output object file is valid.

ACTION: Rewrite the MEMORY definition. Ensure that the item referred to in the ALLOCATE construct (a segment, table, or task) has attributes that match those of the range named; then reinvoke BLD386.

WARNING 266: ADDRESS ALREADY ASSIGNED

MEANING: BLD386 does no forward referencing. In MEMORY definition, the ALLOCATE construct refers to an item identified by a BASE. If the MEMORY definition follows a SEGMENT or TABLE definition where the base address of this item was assigned, then the base cannot be reassigned in the MEMORY definition. If the SEGMENT or TABLE definition follows the MEMORY definition, then the base address is already assigned in the MEMORY definition and cannot be reassigned according to the SEGMENT or TABLE definition.

EFFECT: BLD386 processing continues. The desired use of the ALLOCATE construct does not take place. The output object file is valid.

ACTION: Rewrite the MEMORY definition or SEGMENT definition lines. Ensure that the item referred to in the ALLOCATE construct precedes the ALLOCATE construct and does not have a base address assigned. Reinvoke BLD386.

WARNING 267: ALIAS IS ABSOLUTE

MEANING: The segment specified by an alias segment name in an ALIAS definition is already absolute.

EFFECT: BLD386 processing continues. The aliasing that was requested does not occur. The output object file is valid.

ACTION: Ensure that aliasing is written so as to avoid the described conflict; then reinvoke BLD386.

WARNING 268: ALREADY AN ALIAS

MEANING: The segment specified by an alias segment name in an ALIAS definition is already an alias of some entity. BLD386 does not allow the same segment to be an alias of two different entities.

EFFECT: BLD386 processing continues. Only the first aliasing is done. The output object file is valid.

ACTION: Ensure that the segment being aliased has not already been aliased. Rewrite the alias to avoid the described conflict; then reinvoke BLD386.

WARNING 269: SEGMENT SIZE REDUCED

MEANING: The exact size of the segment specified by the LIMIT construct (that is, without the + prefix) is less than the actual size of the specified segment.

EFFECT: BLD386 processing continues and pads the segment size. BLD386 issues this warning to notify you of the situation. The output object file is valid.

ACTION: If the specified limit value is correct, ignore the warning. If the specified value is not correct, change the limit and reinvoke BLD386.

ERROR 270: ILLEGAL SPECLN VALUE - USING 0

MEANING: The number specified as SPECLN was illegal, maybe not numeric.

EFFECT: No SPECLN is specified for the current segment

ACTION: Reenter SPECLN in correct format; then reinvoke BLD386.

ERROR 271: MISMATCH (USE16<>USE32) BETWEEN ENTRY SEGMENT AND GATE

MEANING: The 80286/80386 attribute of a gate was specified differently from that of the code segment that has the entry point name.

EFFECT: BLD386 processing continues. Attributes of gate and segment are unchanged. The output object file is valid.

ACTION: Correct either gate or segment attribute; then reinvoke BLD386.

ERROR 272: CANNOT SPECIFY USE16 OR USE32 FOR TASK GATE

MEANING: The USE16/USE32 attribute can be specified for call, trap, and interrupt gates but not for a task gate because a task gate has no meaning.

EFFECT: BLD386 processing continues. The USE attribute is ignored. The output object file is valid.

ACTION: Reenter GATE definition without offending the attribute; then reinvoke BLD386.

ERROR 273: CANNOT SPECIFY VIRTUALMODE WITHOUT BOOTLOAD OPTION

MEANING: The VIRTUALMODE parameter in TASK definition is only legal in conjunction with the BOOTLOAD control in the invocation line.

EFFECT: The request is ignored.

ACTION: Resubmit BLD386 with the correct invocation line.

ERROR 274: BASE INCOMPATIBLE WITH ALIGNMENT

MEANING: If a base address of a segment is specified, and alignment is specified, they must agree with each other; e.g., a word-aligned segment must be based on an even address etc.

EFFECT: BLD386 processing continues. The alignment is ignored. The output object file is valid.

ACTION: Reenter SEGMENT definition with base or align corrected; then reinvoke BLD386.

ERROR 275: TSS MUST BE INPAGE ALIGNED

MEANING: If a base address is specified for TSS, it must be on a paragraph boundary and not cross a page boundary

EFFECT: BLD386 processing continues. TSS is left on same base but incorrectly aligned. The output object file is not valid.

ACTION: Reenter base correctly; then reinvoke BLD386.

ERROR 276: CANNOT DO PAGING

MEANING: (a) PAGING definition in build file can only be processed if invocation line controls BOOTLOAD and PAGETABLES are in effect. (b) Memory to be paged was not defined either via MEMORY definition or PAGING definition.

EFFECT: BLD386 processing continues. PAGING definition is ignored. The output object file is valid.

ACTION: Correct invocation line or build file; then reinvoke BLD386.

ERROR 277: LOWER BOUND HIGHER THAN UPPER BOUND

MEANING: When a memory range was defined, either for reserved or paged memory, the lower bound was specified at a higher value than the upper bound.

EFFECT: BLD386 processing continues. Range definition is ignored. The output object file is valid.

ACTION: Correct build file; then reinvoke BLD386.

ERROR 278: LOWER ADDRESS NOT PAGE ALIGNED

MEANING: When a memory range was defined for paging, its lower bound was not page-aligned. Note that the address printed cannot appear in the build file line reported, but in a previous definition of a range.

EFFECT: BLD386 processing continues. Range definition ignored. The output object file is valid.

ACTION: Correct build file; then reinvoke BLD386.

ERROR 279: IRRECOVERABLE SYNTAX ERROR

MEANING: This fatal error occurred because there were errors that the recovery mechanism could not overcome when parsing the build file.

EFFECT: BLD386 processing is aborted, and control is returned to the operating system.

ACTION: Correct the build file; then reinvoke BLD386.

WARNING 280: CANNOT SPECIFY NOT CREATED WITHOUT BOOTLOAD OPTION

MEANING: The NOT CREATED parameter in TABLE definition is only legal in conjunction with the BOOTLOAD control in the invocation line.

EFFECT: The request is ignored.

ACTION: Resubmit BLD386 with the correct invocation line.

ERROR 281: UPPER ADDRESS NOT ALIGNED TO OFFFH

MEANING: When a memory range was defined for paging, its upper bound was not aligned to one less than page boundary. Note that the address printed cannot appear in the build file line reported, but in a previous definition of a range.

EFFECT: BLD386 processing continues. Range definition is ignored. The output object file is valid.

ACTION: Correct build file; then reinvoke BLD386.

ERROR 282: THIS BLOCK OVERLAPS PREVIOUSLY DEFINED BLOCK

MEANING: When defining a memory range for reserved or paged memory, this range overlaps a range already defined.

EFFECT: BLD386 processing continues. Range definition is ignored. The output object file is valid.

ACTION: Reenter range definitions without overlap; then reinvoke BLD386.

ERROR 283: NO RANGE WITH THIS NAME

MEANING: When specifying by name a memory range for paging, this name was not previously defined in memory section.

EFFECT: BLD386 processing continues. The range is ignored. The output object file is valid.

ACTION: Reenter the range with correct name or insert the MEMORY definition with that name; then reinvoke BLD386.

ERROR 284: THIS RANGE WAS NOT INCLUDED IN PAGE TABLES

MEANING: This error occurred because an attempt was made to set bits for a range not previously defined as present in the BITSETTING construct of the PAGING definition.

EFFECT: BLD386 processing continues. BITSETTING is ignored. The output object file is valid.

ACTION: Correct bounds of BITSETTING; then reinvoke BLD386.

ERROR 285: SPECLN > SLIMIT, TRUNCATED

MEANING: You cannot specify a SPECLN that is greater than the size of the segment. This error can also occur if attributes of a segment are changed in subsequent SEGMENT definitions; e.g., the limit is reduced so it is less than SPECLN; before the SEGMENT definition this was greater than SPECLN.

EFFECT: BLD386 processing continues. SPECLN is reduced to SLIMIT+1. The output object file is valid.

ACTION: Correct SEGMENT definition; then reinvoke BLD386.

ERROR 286: CANNOT PLACE DIRECTORY ADDRESS IN EXEC. ONLY

MEANING: The LOCATION parameter of the PAGING definition specified a public name in a segment whose attribute is execute-only, i.e., forbidden to write in the address of the page directory.

EFFECT: BLD386 processing continues. LOCATION is ignored. The output object file is valid.

ACTION: Either change the access byte of the segment holding this public name or delete the LOCATION parameter; then reinvoke BLD386.

ERROR 287: PAGETABLES CAN ONLY BE DEFINED ONCE

MEANING: This error occurred because the PAGING definition was used more than one time.

EFFECT: BLD386 processing aborts.

ACTION: Rewrite the build file so that the PAGING definition is used only once.

ERROR 288: DIRECTORY/PAGE TABLE SEGMENT NOT R/W, PAGEALIGN, AND EXPAND UP

MEANING: You cannot add a page directory or tables to a segment that does not have Read/Write access, page aligned, and not expanddown attributes.

EFFECT: BLD386 processing continues. The PAGING definition is ignored. The output object file is valid.

ACTION: Either change the access byte of segment, or place directory/tables in a different segment; then reinvoke BLD386.

ERROR 289: CANNOT QUALIFY NAME OF NEW SEGMENT

MEANING: When specifying the name of a new segment for page directory or tables, you cannot specify a module name together with a segment name.

EFFECT: BLD386 processing continues. The PAGING definition is ignored. The output object file is valid.

ACTION: Reenter the segment name with a legal name; then reinvoke BLD386.

ERROR 290: MODULE ALREADY EXISTS

MEANING: When defining a new segment name via CREATESEG definition, a segment with this name was already read in from input object files.

EFFECT: BLD386 processing continues. The CREATESEG definition is ignored. The output object file is valid.

ACTION: Define the new segment with a unique name; then reinvoke BLD386.

ERROR 291: SYMBOL NAME ALREADY EXISTS

MEANING: In CREATESEG definition, when defining a symbol name for a new segment, an existing public name was found with the same name.

EFFECT: BLD386 processing continues. The SYMBOL construct was ignored. The output object file is valid.

ACTION: Define the new segment with a unique name; then reinvoke BLD386.

WARNING 292: NO LONGER SUPPORTED

MEANING: The SEGMENT definition parameter [NOT] EXPANDABLE is no longer supported by BLD386.

EFFECT: BLD386 processing continues. The [NOT] EXPANDABLE parameter is ignored. The output object file is valid.

ACTION: Delete from build file; then reinvoke BLD386.

ERROR 294: EXPORT FILE NAME MISSING

MEANING: In EXPORT definition, the name of the output file containing exported entities (comes after #) was not supplied.

EFFECT: No export file can be created.

ACTION: Correct the build file; then reinvoke BLD386.

ERROR 295: ITEM ALREADY ASSIGNED TO SLOT

MEANING: The item has already been assigned a slot in a descriptor table.

EFFECT: BLD386 processing aborts. The object file is invalid.

ACTION: Correct the build file by ensuring that the same item is not assigned to a descriptor table more than once. If you would like more than one descriptor for the same item, refer to the ALIAS definition.

ERROR 296: AMBIGUOUS NAME

MEANING: This error occurs because more than one module contains the segment referred to in a SEGMENT or EXPORT definition.

EFFECT: The segment is regarded as not found; thus this message is followed by Warning 225.

ACTION: Correct the build file by replacing the segment name by its fully qualified name (including the module name).

ERROR 297: NAME REFERS TO BSS SYMBOL

MEANING: In CREATESEG definition, when defining a symbol name for a new segment, an external definition for a BSS symbol was found with the same name.

EFFECT: The SYMBOL construct was ignored.

ACTION: Check the source program that contains the external definition. Remember that CREATESEG definition is intended to create stacks, not data areas.

WARNING 298: USERREAL COMBINATION INCORRECT

MEANING: The USERREAL parameter in SEGMENT definition was used together with one or more of the following:

- a) NOBOOTLOAD control in the invocation line
- b) USE32 segment (from input or via build language)
- c) Alignment not paragraph, inpage, or page, and no base assigned
- d) Base assigned that is not paragraph-aligned

EFFECT: USERREAL is ignored.

ACTION: Correct combination and resubmit BLD386.

ERROR 299: 80376 DOES NOT SUPPORT USERREAL SEGMENTS

MEANING: The 80376 does not support USERREAL segments. A USERREAL keyword was found in the build file SEGMENT definition.

EFFECT: BLD386 continues processing but the object file will be invalid.

ACTION: Correct the build file; then reinvoke BLD386.

ERROR 500: 80376 DOES NOT SUPPORT USE16 GATES

MEANING: USE16 gates are not supported on the 80376. A USE16 gate was found in the build file definition.

EFFECT: BLD386 continues processing but the object file will be invalid.

ACTION: Correct the build file; then reinvoke BLD386.

WARNING 501: MEMORY RANGES WILL OVERLAP ON THE 80376

MEANING: The build file MEMORY definition contains a reserve definition or range definition which specifies virtual address memory greater than physical memory. Because the address bus is less than 32 bits, virtual addresses that are greater than the bus size will wrap to low memory. This could overlay segments that are in low memory. Note that if text is overlapped by text from another segment, BLD386 will also issue Warning 402. See the MOD376 control in Chapter 4.

EFFECT: BLD386 continues processing but the object file could be invalid.

ACTION: Correct the build file, if needed; then reinvoke BLD386.

ERROR 502: 80376 DOES NOT SUPPORT VIRTUAL MODE IN TASK DEFINITION

MEANING: The TASK definition in the build file specified VIRTUALMODE. The 80376 does not support virtual mode.

EFFECT: The request is ignored.

ACTION: Correct the build file; then reinvoke BLD386.

ERROR 503: 80376 DOES NOT SUPPORT USE16 CODE SEGMENTS

MEANING: The 80376 does not support USE16 code segments. The USE16 keyword was found in the build file code SEGMENT definition.

EFFECT: BLD386 continues processing but the object file will be invalid.

ACTION: Correct the build file; then reinvoke BLD386.

ERROR 504: 80376 DOES NOT SUPPORT PAGING

MEANING: The 80376 does not support paging. The PAGING definition was specified in the build file.

EFFECT: BLD386 processing continues. The PAGING definition is ignored. The output object file is valid.

ACTION: Correct invocation line or build file; then reinvoke BLD386.

WARNING 505: *id* IDENTIFIER HAS NOT BEEN ADDED TO RANGE

MEANING: BLD386 does no forward referencing. In MEMORY definition the ALLOCATE construct contains the * option (meaning include all tasks, segments, or tables). The first reference to the identifier follows the MEMORY definition. This can be a named task, an LDT, or a segment created with the CREATESEG definition.

EFFECT: Since the builder does not know about the identifier when it processes the MEMORY definition, this identifier is not added to the range.

ACTION: Ensure that all definitions referred to in the MEMORY definition ALLOCATE construct precede the MEMORY definition ALLOCATE construct. Reinvoke BLD386.

WARNING 506: *id* IDENTIFIER HAS NOT BEEN DEFINED

MEANING: BLD386 does no forward referencing. In MEMORY definition, the ALLOCATE construct refers to an item identified by name. However, the item was either not specified in the build file, does not exist in the input object module, or follows the MEMORY definition in the build file. Certain tables (GDT and IDT) are known to the builder before build file processing. However, BLD386 does not know what LDT, task state segments, or CREATESEG definitions the user wants until encountered in the build file or the build file processing is complete.

EFFECT: BLD386 continues processing. The desired use of the ALLOCATE construct does not take place. The output object file is valid.

ACTION: If the item referred to in the ALLOCATE construct (a segment, table, or task) does not exist in the build file or input object file, then be sure to specify it. If the item referred to in the ALLOCATE construct follows the ALLOCATE construct, change the build file so that it is placed before the ALLOCATE construct. Reinvoke BLD386.

ERROR 507: PHANTOM SEGMENTS MUST HAVE THE SAME BASE ADDRESS

MEANING: The user has specified the FLAT control in the invocation line and has based the phantom segments in the build file. However, `__phantom_code__` and `__phantom_data__` were not given the same base address.

EFFECT: Build processing continues. The base addresses of both `__phantom_code__` and `__phantom_data__` are set to zero. The object file is valid.

ACTION: Correct the build file and give the phantom segments the same base addresses, or let them both default to base zero.

C.4 Internal Processing Exceptions

Should you encounter an error message with the following format, contact your Intel representative. A fatal error has occurred that is internal to your system or software that cannot be corrected by a user.

* * * ERROR 3xx: INTERNAL PROCESSING ERROR, *message*

Where:

3xx is the error number.

message is a string that contains the text of the message.



Appendix D

Master List of Reserved Words

The following list contains all reserved words, keywords, and abbreviations for the build language:

Word	Abbr.	Word	Abbr.
ALIAS	AI	P	
ALIGN	AN	PAGE	
ALLOCATE	AL	PAGED	PD
AT		PAGETABLES	PT
BASE	BA	PAGING	PA
BITSETTING	BS	PARAGRAPH	PH
BYTE	BY	[NOT] PRESENT	[NO] PS
CALL	CA	QWORD	QW
CODE	CO	RAM	
[NOT] CONFORMING	[NO] CF	RANGE	RN
[NOT] CREATED	[NO] CD	[NOT] READABLE	[NO] RA
CREATESEG	CS	RESERVE	RS
DATA	DT	RESET	
[NOT] DEBUGTRAP	[NO] DB	ROM	
DPL		RW	
DRESET		SEGMENT	SM
DSET		*SEGMENTS	
DWORD	DW	SET	
END	DN	SPECLN	SP
ENTRY	ET	STACKS	ST
[NOT] EXPANDDOWN	[NO] ED	SYMBOL	SB
EXPORT	EO	TABLE	TB
GATE	GA	*TABLES	
GDT		TASK	TA
IDT		*TASKS	
[NOT] INITIAL	[NO] II	TRAP	TR
INPAGE	IN	UD1	
[NOT] INTENABLED	[NO] IE	UD2	
INTERRUPT	IT	UD3	
IOPRIVILEGE	IP	US	

Word	Abbr.	Word	Abbr.
LDT		USE16	
LIMIT	LI	USE32	
LOCATION	LA	USEREAL	USERL
MEMORY	MO	[NOT] VIRTUALMODE	[NO] VM
NOT	NO	WC	
OBJECT	OJ	WORD	WO
		[NOT] WRITABLE	[NO] WA

Appendix E

Master List of Controls and Abbreviations

The following list contains all the invocation controls and their abbreviations:

Word	Abbr.	Word	Abbr.
[NO]BOOTLOAD	[NO]BL	[NO]MAP	[NO]MA
BOOTSTRAP	BS	MOD376	M376
[NO]BUILDFILE	[NO]BF	MOD386	M386
CONTROLFILE	CF	[NO]OBJECT	[NO]OJ
[NO]DEBUG	[NO]DB	PAGETABLES	PT
[NO]ERRORPRINT	[NO]EP	PRINT	[NO]PR
[NO]FILL	[NO]FI	RELDISC	RD
FLAT	FL	TITLE	TT
[NO]LIST	[NO]LI	[NO]TYPE	[NO]TY
		[NO]WARNINGS	[NO]WA



This appendix is a guide to installing the RL386 software.

F.1 Installation Media

The RL386 software is packaged in a single 1600-BPI magnetic tape containing the VAX-RL386 application programs in VMS BACKUP format.

F.2 Execution Environment

The RL386 software executes on Digital Equipment Corporation VAX-11 models 730, 750, 780, 782, 785, and compatible models running under the VMS operating system release 4.1 or above.

F.3 Installation Requirements

Ensure that the following requirements are met before starting the VMSINSTAL utility:

- SETPRV privileges, or CMKRNL, WORLD, and SYSPRV privileges
- 2500 blocks of free disk space (minimum, during installation)
- 1300 blocks of disk space (after installation)
- 512K bytes of physical memory
- Approximate installation time: 15 minutes

F.4 Installation Procedure

The VAX-based RL386 applications are installed using the VMSINSTAL utility. The installation procedure is displayed step-by-step on the console. The system prompts the user at each step and waits for user input. The user enters a response and presses <cr>. If there is a question about any step, the user types a question mark (?); an explanation is then displayed on the console.

1. Log onto the system manager's account.
2. Ensure that the VMS system is release 4.1 or later.
3. Set the default directory to SYSS\$UPDATE.
4. Invoke the VMSINSTAL procedure giving the name of the product (RL386). VMSINSTAL prompts for the name of the tape drive (for example, MSA0).
5. A sign-on message is then displayed.
6. Mount the tape on a tape drive and put the drive on line.
7. A prompt then appears for the name of the directory where the RL386 images are to be placed.
8. VMSINSTAL procedure prompts the user on whether or not to run the Installation Verification Procedure (IVP).
9. If installation has been successful, a message stating so appears, and the files listed in section "RL386 Installation Files" in this appendix are on the target disk.

When the above steps are completed, RL386 tools are ready to use.

Listed below is a sample installation session on the VAX. Underlined text represents user input.

```
Username: SYSTEM <cr>
Password:            <cr>

$ SET DEFAULT SYSS$UPDATE <cr>
$ @VMSINSTAL RL386 <cr>
```

VAX/VMS Software Product Installation Procedure

It is 2-AUG-1988 at 16:31.

Enter a question mark (?) at any time for help.

%VMSINSTAL-W-DECNET, Your DECnet network is up and running.

* Do you want to continue anyway [NO]? y

* Are you satisfied with the backup of your system disk [YES]? y

* Where will the distribution volumes be mounted [MSA0:] <tape-device>

Please mount the first volume of the set on <tape-device>

* Are you ready? y

%MOUNT-I-MOUNTED, R386KT mounted on <tape-device>

The following products will be processed:

RL386 V1.3

Beginning installation of RL386 V1.3 at 16:33

% VMSINSTAL-I-RESTORE, Restoring product saveset A...

```
*-----*
* Installation Command Procedure for *
*          VAX/VMS RL386 V1.3          *
*-----*
```

* Do you want to purge files replaced by this installation [YES]? y

This kit contains an Installation Verification Procedure to verify the correct installation of the VAX/VMS RL386 tools.

* Do you want to run the IVP after the installation [YES]? y

* Enter where the RL386 V1.3 images directory INTEL386 should be created: <device-name>

%VMSINSTAL-I-SYSDIR, This product creates system directory [SYSHLP.EASE].

If you intend to execute this layered product on other nodes in your VAXcluster, and you have the appropriate software license, you must prepare the system-specific roots on the other nodes by issuing the following command on each node (using a suitable privileged account):

```
$ CREATE /DIRECTORY SYS$SPECIFIC:[SYSHLP.EASE]
```

VAX/VMS RL386 V1.3 Installation completed successfully

%VMSINSTAL-I-MOVEFILES, Files will now be moved to their target directories...

```
*-----*
* Installation Verification Procedure for *
*           VAX/VMS RL386 V1.3           *
*-----*
```

TEST MAP386

VAX/VMS 80386 MAPPER V1.1VX
Copyright 1986, Intel Corporation

PROCESSING COMPLETED. 0 WARNINGS, 0 ERRORS

VAX/VMS MAP386 Installation Verification completed successfully

TEST LIB386

VAX/VMS 80386 LIBRARIAN, V1.1VX
Copyright 1986, Intel Corporation

TARGET LIBRARY: NORMALIZED_LIBRARY 0000 08/02/88 11:49:08

3 MODULES ADDED, 1 MODULES DELETED

**A LIBADD.LIB

**D ACS

**L TO TMP.LIS P

**Q E

TMP.LIB, NORMALIZED_LIBRARY 0000 08/02/88 11:49:08

3 MODULES ADDED, 1 MODULES DELETED

VAX/VMS LIB386 Installation Verification completed successfully

TEST BLD386

VAX/VMS 80386 SYSTEM BUILDER, V1.4VX
Copyright 1986, 1988 Intel Corporation

PROCESSING COMPLETED. 0 WARNINGS, 0 ERRORS

VAX/VMS BLD386 Installation Verification completed successfully

TEST BND386

VAX/VMS 80386 BINDER, V1.3VX
Copyright 1986, Intel Corporation

PROCESSING COMPLETED. 0 WARNINGS, 0 ERRORS

VAX/VMS BND386 Installation Verification completed successfully

Installation of RL386 V1.3 completed at 16:42

VMSINSTAL procedure done at 16:43

F.5 RL386 Installation Files

After a successful installation, the following files will be on the target disk:

I386__EXE:MAP386.EXE
I386__EXE:BLD386.EXE
I386__EXE:LIB386.EXE
I386__EXE:BND386.EXE

The installation procedure also creates the file RL386START.COM in the system manager's directory SYSS\$MANAGER. This file should be part of the system start-up.

F.6 RL386 Commands and the Help Facility


The DCL commands that activate the images are integrated with the VMS system commands. The help text is inserted into the system help library.

F.7 Ease of Use Kit

The following Ease of Use Kit files are installed in the directory
SYS&ROOT: [SYSHLP.EASE]:

APPLIC.ASM;1
APPLIC.BLD;1
BITCNT.C;1
CSTART.A38;1
FLAT.A38;1
FLAT.BLD;1
FLAT.COM;1
FLATSIM.A38.1
FLATSIM.BLD;1
FLATSIM.COM;1
INTPROC.ASM;1
INTRPT.A38;1
PROT.COM;1
PROTFLAT.A38;1
README.TXT;1
REVR.S.C;1
SHOWBITC.INC;1
SHOWREVR.INC;1
SIMPLE.C;1
SIMPLE.INC;1
SYS.ASM;1
SYS.INC;1
SYS1.INC;1





80376 microprocessor

the 80376: an advanced, high-performance microprocessor in the Intel386™ family capable of running 32-bit protected-mode software.

80386 Binder

the 80386 program development utility used to link modules, combine segments, and create a single-task loadable output module. This product is part of the 80386 Development Package.

80386 Librarian

the interactive 80386 program development utility that enables you to create, manipulate, and examine 80386 library files. This product is part of the 80386 Development Package.

80386 Macro Assembler

the assembler used to produce linkable object modules that are executable on 80386 and 80387 processors in protected mode. This product is part of the 80386 Development Package.




80386 Mapper

the 80386 program development utility that produces a cross-reference map for public or external symbols from information in a loadable or bootloadable module. This product is part of the 80386 Development Package.

80386 microprocessor




the 80386: an advanced, high-performance microprocessor with capabilities for multiple-user and multitasking systems. The processor has built-in memory protection for isolating memory space from task to task. The processor can execute 16-bit 8086 or 80286 code. Built-in features implement many details of a paged software system. The 80387 Numeric Processor Extension provides high-speed floating-point capabilities.



80386 Software Development Package

the group of software products that provides the capabilities to develop simple or multitasking protected-mode systems for the Intel386 family processors.

80386 System Builder	BLD386, the configuration utility for 80386 protected mode systems. This product is part of the 80386 Software Development Package.
absolute address	the physical location in which code or data resides in memory. In protected virtual address mode the 80386 processor supports 32-bit absolute addresses.
absolute object code	code or data to which absolute addresses have been assigned.
access rights	the segment attributes that describe how a segment can be accessed by other segments. Access rights for stack and data segments include read-only and read-write. Access rights for code segments include execute-only, executable and readable, and conforming.
alias	a new name assigned to represent the same item as an existing name. The new name can exist in descriptor table slots, and can allow access to the same absolute memory locations while using different attributes for those locations, if desired.
aliasing	establishing aliases for segments, tables, or tasks.
ASM386	the 80386 Macro Assembler mnemonic.
attribute	one of the entities defined by a descriptor: base address, limit, and access byte parameters. Segment usage violating an attribute causes an exception or interrupt.
base address	the 32-bit address at which a segment starts.
binder	see 80386 Binder.
BLD386	the 80386 System Builder mnemonic.
BND386	the 80386 Binder mnemonic.
bootloadable module	a module that contains absolute object code in a very simple format to expedite the loading of cold-start modules.

bootstrap	the initial instruction or program executed after system reset. The program may be located permanently in ROM or loaded with a bootstrap loader.
 bootstrap loader	a system that loads the bootstrap program from auxiliary storage onto a computer.
BSS	block symbol storage.
builder	see 80386 System Builder.
build file	a file that contains a build program.
build file listing	the section of a BLD386 print file that contains an annotated version of the build program.
build language	a declarative language used to create definitions for system data structures. These definitions are contained in a build program.
build program	a program that contains one or more build language definitions of segment descriptors, gates, TSSs, descriptor tables, elements to export, and memory reservations.
 call gate	a gate used to transfer control to more privileged code in a task.
checksum byte	the last byte in a loadable or bootloadable module created by the 80386 utilities. This byte is the complement of the sum of all the preceding bytes in the object module, including the header.
combine name	the name of a segment.
combine type	one of the characteristics associated with segments. Combine types include normal, stack, and data/stack combined (DSC).
conforming segment	a code segment that can be shared by programs that execute at different privilege levels. No gate is needed to access a conforming segment.
 contention	the apparent overlap of two entities in the same address space.

control file	a BLD386 input file that contains the file names of linkable input modules and/or controls.
control-transfer descriptors	call gates, task gates, interrupt gates, and trap gates.
CPL	current privilege level.
current privilege level	a task's privilege level at any specific instant, indicated by the lower 2 bits of the CS register.
data/stack combined	the combine type associated with segments created when normal data and stack segments that have the same combine name are combined. The segment is expanddown.
debug information	symbolic information in an object module that is used by debuggers.
default value	a value assigned automatically if no explicit specification is given otherwise.
descriptor	an 8-byte entity that defines the use of memory in an 80386 protected virtual address mode system. Descriptors include segment descriptors and system control descriptors.
descriptor installation	assigning individual descriptors to slots (locations) in descriptor tables.
descriptor internal name	a 32-bit value that defines the location of a descriptor in a bootloadable or loadable object module.
descriptor privilege level	the privilege level defined in the descriptor for a segment or in a gate.
DPL	descriptor privilege level.
DSC	data/stack combined.
dynamically loadable module	an executable object module that can be dynamically loaded into an Intel386™ family-based system with an operating system running.
dynamic system	a system that has a memory configuration defined at load time and/or run time.

entry point	the destination selector and offset for call, interrupt, and trap gates; or the destination selector for task gates.
EO	execute-only.
ER	executable and readable.
error condition	a condition that causes BLD386 to issue a warning, error, or fatal error message.
error message	a BLD386 message that flags a condition that causes the output object module(s) to be undefined.
executable file	a file containing a loadable or bootloadable module.
executable and readable segment	a code segment that can be read.
executable segment	a code segment.
execute-only segment	a code segment that cannot be read.
expanddown segment	a nonexecutable segment that has a limit that can be extended toward lower-order addresses at run time.
exportation	a process by which a system interface is made available to application and system programs. Gates and segments providing access to system services (such as input/output management and exception handling) are placed with the EXPORT definition into a linkable module. This module is used when configuring loadable tasks that depend on the exported services.
export file	a linkable file created by BLD386 that contains modules and/or gates defined in the build language.
export module	a module within an export file into which BLD386 places exported information.
external reference	reference to a symbol, procedure, or location defined as public in another module.

fatal error message	a BLD386 message that indicates an encountered error condition that prohibited the processing from being completed.
file name	a character string recognized by the operating system to identify a file, including the device-name and directory where necessary.
flat model	a model of memory where physical addresses correspond 1:1 with linear addresses. For the Intel386 family, a bootloadable module can be configured in flat mode; i.e., there is one code segment overlapped with one data segment each with addressing from 0 to 4 gigabytes-1.
gate	a descriptor used to regulate access to code at a higher privilege level (call gate), code in a different task (task gate), or interrupt service routines (interrupt or trap gates).
GDT	global descriptor table.
global descriptor table	a table that houses descriptors that are available to all tasks. This table can contain as many as 8190 descriptors of the following types: segment descriptors, TSS descriptors, LDT descriptors, call gates, and task gates.
granularity	the characterization of a segment's limit. If the granularity is 0, the limit is expressed in bytes. If the granularity is 1, the limit is expressed in units of 4K bytes.
identifier	a name used in build language specification; it has as many as 40 characters and refers to a build program, a gate, an LDT, a module, a public symbol, a segment, or a task.
IDT	interrupt descriptor table.
illegal access	an attempted code or data access that causes an exception condition because it violates hardware protection mechanisms. Protection is implemented by enforcing segment access rights and privilege and descriptor usage rules.
index	a position or slot in which a descriptor is to be installed in a descriptor table.

initialization information	register values that need to be established before task execution begins.
internal name	a fixed-length name that has a single module as its scope. Segments, externals, types, gates, and descriptors all have internal names.
interrupt descriptor table	the descriptor table that houses up to 256 interrupt, trap, and/or task gates and is used to regulate access to routines for handling interrupt and exception conditions. Slots numbered 0 through 31 correspond to Intel-defined or Intel-reserved interrupts.
interrupt gate	a descriptor that points to an interrupt service routine and whose use disables interrupts; see also trap gates.
intersegment reference	the reference to a location in a segment different from the segment that contains the reference (sometimes called FAR in source languages).
intra segment reference	the reference to a location in the same segment as the segment that contains the reference (sometimes called NEAR in source languages).
LDT	local descriptor table.
LDT descriptor	a descriptor that refers to an LDT.
LDT selector	a selector that is installed in a TSS and points to a particular LDT.
LIB386	the 80386 Librarian mnemonic.
library	an object library, consisting of one or more sets of linkable modules. Object libraries are produced by LIB386 and reside in library files.
library file	a file that contains a collection of linkable object modules indexed (at least) by module names.
limit	the segment attribute that defines the offset of the last byte in the segment.
linkable files	a file containing one or more linkable modules.

linkable module	an object module created by 80286 or 80386 translators, BND386, or BLD386; this module serves as input to either 80386 utility. A linkable module requires further processing before it can be executed.
linking	combining segments from one or more linkable input modules and resolving references between modules; BND386 provides linking.
loadable module	see dynamically loadable module. A module to load onto a running system.
loader	a program that establishes an initial memory configuration for a task prior to execution.
local descriptor table	a table that houses up to 8191 descriptors that can be private to a task. An LDT can contain only segment descriptors, task gates, and call gates. BLD386 can be used to define slots 0 through 8190 of an LDT.
MAP386	the 80386 Mapper mnemonic.
module	an named entity which can contain one or more other entities with a cohesive purpose.
normal	the combine type associated with segments that contain only code or data.
object module	a module which is executable.
object module format	the structure of an object module.
offset	a byte address in a segment.
paging	a technique for redefining a linear address as a real address.
physical segment	a contiguous piece of memory that cannot exceed 64K bytes in length for USE16 attribute or 4G bytes for USE32 attribute.
privilege hierarchy	the aspect of the 80386 protection scheme that provides up to four different segment access levels.



privilege level

the attribute that ranges from 0 through 3 and controls the use of privileged instructions as well as access to descriptors and their segments. Access in a protected-mode system uses three kinds of privilege levels: current privilege level, descriptor privilege level, and requested privilege level.

privilege rules

rules that govern how and when a task can access a segment. These rules use the following parameters: the type of segment to be accessed, the instruction used, the type of descriptor used, the current privilege level, the requested privilege level, and the descriptor privilege level.

protected mode

the mode of operation of the 80386 processor that provides protection.

protection

80386 mechanism that ensures that code and data segments are insulated from improper usage, and that the critical CPU execution state control instructions are properly implemented.



public

(1) a symbol or procedure available for intersegment or intrasegment references. (2) a kind of ASM386 segment.

readable segment

a code segment that can be read.

read-only segment

a data segment that can only be read.

read-write segment

a data or stack segment that can be read from and written to.

real address

physical address that specifies an absolute location in memory. In 80386 protected mode, the real address has 32 bits.

reference resolution

the process by which public definitions are paired with external references.




relocatable information




code or data that has a location defined at load or run time.



requested privilege level

the privilege level defined as the least significant two bits of a selector. It is used with the current privilege level to establish the privilege levels a task can access.

RO	read-only.	
RPL	requested privilege level.	
run time	when the program executes.	
RW	readable and writable; read/write.	
section	a structural subdivision of an object module: object modules are divided into a header and one or more sections.	
segment	(1) a contiguous piece of memory. (2) code and/or data which will be loaded into a contiguous piece of memory.	
segment attributes	see attributes.	
segment base	the 32-bit address at which a segment begins.	
segment descriptor	a descriptor that refers to code, stack, and data segments in a program.	
segment map	the BLD386 print file section that provides information for all segment descriptors in the output module.	
selector	an index into a descriptor table. GDT and LDT selectors are 16-bit pointers that index the GDT and LDTs, respectively; IDT selectors are 8-bit vectors into an IDT.	
slot	a location in a descriptor table.	
small model	a program memory model where text and data reside in the same segment (impure), or in different segments (pure), and pointers and integers are 16 bits. Other memory models include middle, large, and huge.	
system data segment descriptors	TSS and LDT descriptors.	
static system	a system that has a total memory configuration prior to loading (statically). A dedicated control system is an example of a static system.	
stack	the combine type associated with stack segments.	

subsystem	a group of related procedures, tasks, and/or modules with a well-defined purpose. It is likely that the entities within the subsystem share data or have a message-passing mechanism.
 symbol	(1) a variable in a module. (2) in BND386 or BLD386, an internal representation of an object module entity.
system builder	see 80386 System Builder.
system building	the configuration of a system, especially the selective definition of system data structures and tasks and the allocation of privilege among segments and descriptors.
system data structures	descriptor tables, segment and system descriptors, and TSSs.
system descriptors	special system data segment descriptors and control-transfer descriptors.
table indicator	in a 16-bit selector, the bit that defines whether the selector points to the GDT or an LDT.
 target	(1) a referenced entity. (2) the host machine onto which a program is loaded.
task	a single, sequential thread of execution. It has an associated processor state and a well-defined address space with specific access parameters. The processor state is defined by the contents of the TSS, whereas the address space and access parameters are defined by descriptors.
task gate	a gate used to transfer control to another task. A task gate points to a TSS.
task state segment	the special system segment that stores initialization and restart values for a task. The TSS saves the entire execution state; e.g., registers, address space, and a link to the previous task.
 text	(1) program code and data. (2) ASCII characters in a file.

token	any continuous string of characters with a unique meaning in a program.
translator	an assembler or compiler.
trap gate	a descriptor that points to an interrupt service routine and does not disable interrupts; see also interrupt gate.
TSS	task state segment.
TSS descriptor	a descriptor that defines and points to a TSS.
type	(1) bits 40-43 in a gate descriptor. (2) bit 42 in a segment descriptor. (3) an attribute associated with a segment.
USE16	a segment with the 32-bit attribute off, limited to 64K bytes (80286 compatible).
USE32	a segment with the 32-bit attribute on, limited to 4G bytes.
USEREAL	a segment which is addressed by paragraph number rather than descriptor number, limited to 64K bytes (8086 compatible).
virtual address	an address that consists of a selector and an offset value.
VIRTUALMODE	a construct that controls the setting of the VM bit in the flags register.
warning message	a BLD386 message that indicates that a user error may have occurred; the output object file is valid.
wrap	the process of losing the carry bit in the computation of an address. The computed address points to a valid location in low memory.
writable segment	a stack or data segment that can be written to.

80286

- Object module, 2-13

80286 binder (see BND286)

80376

- Address bus, 4-33

- Gate, 2-5, 2-9, 3-15, 3-16

- Maximum physical address, 4-33

- Paging, 1-8, 1-14, 2-10

- USREAL attribute, 3-36

- USE32 attribute, 2-3

- Virtual mode, 3-58

80386

- Object module, 2-13

- Reset, 4-16

80386 binder (see BND386)

80386 librarian (see LIB386)

80386 mapper (see MAP386)

80386 system builder (see BLD386)

8086

- Paragraph number, 2-3, 3-36

- USREAL attribute, 2-3

A

Absolute address, 2-3, 2-5, 4-13, 4-14

- Overriding, 4-13

Address assignment, 2-3, 2-5, 4-28

- Default, 2-5, 3-23

Address calculation, 1-5

Address translation, 1-8, 1-9

ALIAS definition, 2-1, 3-7 3-9

- Example, 3-9

Alias, 1-13, 2-6, 2-12, 3-7, 3-8, 3-43

- Creation, 2-12

- Items of different sizes, 3-8

ALIGN syntax, 3-33

Alignment, 3-38, 3-41

- Default, 2-5

- Overriding, 2-5

- Page, 3-28, 3-29

- Paragraph, 2-5, 3-36

ALLOCATE syntax, 3-21

Application program, 1-13

ASM386 CALL instruction

- Access rights, 3-56

ASM386 JMP instruction

- Access rights, 3-56

ASM386 LGDT instruction, 3-46

ASM386 LIDT instruction, 3-46

AT syntax, 3-33, 3-43, 3-52

B

Base address, 1-5, 1-8, 1-9

BASE syntax, 3-33, 3-43, 3-52

BITSETTING syntax, 3-27

BLD386, 1-1, 3-1

- Continuation character, 4-3

- DOS syntax, 4-1

- Invocation, 2-1

- VMS syntax, 4-2

- Sign-off message, 4-4, 4-5, 4-24

- Sign-on message, 4-4

Block symbol storage, 2-2, 2-3, 4-25

BND286, 2-13, 5-1

BND386, 1-1, 2-13, 5-1

[NO]BOOTLOAD control, 2-1, 2-2, 4-13

- Examples, 4-14

Bootloadable module, 1-2, 2-5, 2-13, 3-8, 3-43, 3-48, 4-13, 4-14, 4-25, 4-27, 4-35

BOOTSTRAP control, 4-15

- And BOOTLOAD control, 4-15

- And MOD376 control, 4-15

- And MOD386 control, 4-15

- Examples, 4-16

Bootstrap loader, 1-2

BSS (see block symbol storage)

Build file, 2-1, 3-1, 4-6, 4-17

Build language, 3-1

- Abbreviations, 3-3, 3-4

- Build program, 3-1, 3-5

Build language (cont.)

- Comment, 3-2
 - Definition, 3-5
 - Delimiter, 3-1, 3-2, 3-3
 - Examples 3-62 3-63
 - Identifier, 3-2, 3-3
 - Syntax, 3-3
 - Keywords, 3-3, 3-4
 - Number, 3-2, 3-3
 - Decimal, 3-3
 - Hexadecimal, 3-3
 - Syntax, 3-3
 - Order, 3-5
 - Reserved words, 3-3, 3-4
 - Token, 3-1, 3-2, 3-3
- Build program, 1-13, 1-14, 2-13, 3-1, 3-5, 4-6, 4-17
- Definition
 - Alias, 1-13
 - Descriptor table, 1-13
 - Examples, 3-62 3-63
 - Exporting, 1-13
 - Gate, 1-13
 - Memory, 1-14
 - Order, 3-5
 - Page directory, 1-14
 - Page tables, 1-14
 - Segment attribute, 1-13
 - Task state segment, 1-14
 - Listing, 2-1, 2-13, 4-31, 5-4
- [NO]BUILDFILE control, 4-17
- Examples, 4-18
- BYTE syntax, 3-33

C

- Call gates with C, 6-21
 - ASM386 startup code, 6-25
 - Build program, 6-26
 - DOS batch file, 6-27
 - Example interface routines, 6-23
 - Stack cleanup, 6-22
- CALL syntax, 3-15
- CODE syntax, 3-52

- Combine name, 2-3, 3-34
- [NOT] CONFORMING syntax, 3-34
- Console, 2-13, 4-4, 4-23, 4-24
 - Fatal error message, 4-4, 4-5
- Contention, 2-12
- Controls, 2-1, 4-6
 - Abbreviations, 4-12
 - Duplicate, 4-6
 - Summary for DOS, 4-9
 - Summary for VMS, 4-11
 - VMS abbreviations, 4-7
- Control file, 1-2, 2-1, 2-13, 4-5, 4-6, 4-19
 - Comment, 4-6
 - In input-list on VMS, 4-5
 - Line continuation character, 4-6
 - Line length, 4-6
 - Line terminator, 4-6
- Control transfer, 1-9, 1-10
 - Descriptor (see also gate), 2-8, 2-9
- CONTROLFILE control, 4-19
 - Example, 4-20
- [NOT] CREATED syntax, 3-44
- CREATESEG definition, 2-1, 2-2, 2-8, 3-10 3-11
 - Example, 3-11

D

- Data structures, 1-2, 1-5, 3-1
- DATA syntax, 3-52
- [NO]DEBUG control, 2-2, 2-12, 4-21
 - Example, 4-22
- Debugging, 1-1
- Debugging information, 2-2, 2-12, 3-12, 3-53, 4-21, 4-28
 - Removing, 2-12
- [NOT] DEBUGTRAP syntax, 3-53
- Descriptor, 1-5, 2-3, 2-7
 - Installing, 2-6
 - Syntax summary, 3-47
 - Relocation information, 2-12, 4-40
 - Table-alias, 2-7, 3-46, 6-3
- Descriptor table, 1-6, 1-8, 1-9, 1-13, 2-1
 - Creation, 2-6

Descriptor table (cont.)

- Entries, 2-6, 3-45

- In LDT and GDT, 3-48

- In LDT and IDT, 3-48

- Omitting from output, 3-48

- Slots, 1-13, 2-6, 3-44

- DPL syntax, 3-15, 3-33, 3-43, 3-52

- DRESET syntax, 3-27

- DSET syntax, 3-27

- DWORD syntax, 3-33

E

- Entry point, 1-12, 1-13, 2-11

- ENTRY syntax, 3-15, 3-43

- EO (see segment, execute-only)

- ER (see segment, execute/read)

Error

- Fatal, 2-13

- Listing, 2-13

- Error message, 4-23, 4-24, 4-44, 5-13

- [NO]ERRORPRINT control, 2-1, 4-4, 4-23

- And [NO]WARNINGS control, 4-23

- Example, 4-24

- Exception handler, 1-10

- Execution environment, 1-5

- [NOT] EXPANDDOWN syntax, 3-34

- EXPORT definition, 2-2, 3-12 3-14

- Example, 3-14

- Export module, 2-13, 5-1

- Exporting, 1-12, 1-13, 2-2, 2-11

- Entry point, 2-13

- For BND386 or BLD386, 2-11

- Gate, 3-13

- External symbol, 2-11, 4-2, 4-3, 4-4, 4-43

F

- Far call sequence, 6-21

- Far return sequence, 6-22

- [NO]FILL control, 2-2, 4-25

- Examples, 4-26

- FLAT control, 2-1, 2-5, 4-27, 6-2

- Example, 4-30

- Flat model, 2-1, 3-48, 4-27, 4-28, 4-29, 6-1, 6-2

- Overriding defaults, 4-27, 4-28

- Phantom segments, 4-28

- Global descriptor table, 4-28

- Simple, 4-28, 4-29

- Example system, 6-20

- Templates for DOS, 6-1

- Batch file, 6-16

- Build program, 6-9

- C startup template, 6-14

- Example print file, 6-18

- Initialization template, 6-3

- Interrupt routines, 6-13

- Forward-referencing, 2-3, 3-5, 3-23, 3-24

G

- Gate, 1-9, 1-12, 1-13, 2-4

- 80286, 2-2, 3-15, 3-16

- 80286 call, 2-5

- 80286 task, 3-19

- 80386, 2-2, 3-15, 3-16

- 80386 call, 2-5

- 80386 task, 3-19

- 8086, 3-15, 3-19

- Automatic creation, 2-5

- Call, 1-9, 2-3, 2-6

- Default, 2-9

- Far call sequence, 6-21

- Far return sequence, 6-22

- D-bit, 3-17

- Interrupt, 1-9, 2-6

- P-bit, 3-17

- Type bits, 3-16

- Task, 1-9, 2-2, 2-6

- Trap, 1-9, 2-6

- Uninstalled, 2-7

- USE16, 3-16

- And MOD376, 3-19

- USE32, 3-16

- USEREAL, 3-19

- GATE definition, 2-2, 2-9, 3-15 3-20

- Defaults, 3-16

GATE definition (cont.)

- Examples, 3-19 3-20

- Syntax summary, 3-18

Gate descriptor, 1-9, 1-10

- Format, 1-10, 3-17

- Type, 1-10

- Word count, 1-10

GDT (see global descriptor table)

GDT syntax, 3-43

Global descriptor table, 1-6, 1-13, 2-1, 2-6, 2-7, 4-14, 4-28

- As writable data segment, 2-7

- Default organization, 3-47

- Entry-list, 3-49

- Null (first) descriptor, 2-7, 3-47

- Overwriting, 2-7

- Relocating references, 2-2

- Simple, 4-28, 4-29

- Table-alias descriptor, 3-47

I

ICE-386™ In-Circuit Emulator, 6-10, 6-27

IDT (see interrupt descriptor table)

IDT syntax, 3-43

[NOT] INITIAL syntax, 3-53

INPAGE syntax, 3-33

Input, 1-2, 2-13, 4-6, 5-1, 5-2

Intel386 family

- Architecture, 1-5

- Utilities, 1-1

[NOT] INTENABLED syntax, 3-53

Interlevel call sequence, 6-21

Interlevel reference, 1-12

Interlevel return sequence, 6-22

Interrupt descriptor table, 1-13, 2-1, 2-6, 4-14

- As writable data segment, 2-7

- Default, 2-7, 3-46

- Overriding, 2-7, 3-46

- Entry-list, 3-49

- Slots, 2-7

- Table-alias descriptor, 3-47

INTERRUPT syntax, 3-15

Interrupt vectors, 2-7

Intersegment reference, 2-4

IOPRIVILEGE syntax, 3-53

L

LDT (see local descriptor table)

LDT syntax, 3-52

LIB386, 1-1

Libraries, 2-1, 2-11, 2-13, 4-2, 4-3, 4-4

Limit, 1-5

LIMIT syntax, 3-33, 3-43, 3-52

Linear address, 1-8, 1-9, 2-2

Linking, 1-1

[NO]LIST control, 2-1, 4-31, 5-4

- Example, 4-31

Loadable module, 1-2, 2-12, 2-13, 3-7, 3-43, 3-48, 4-25, 4-35, 4-40

Local descriptor table, 1-6, 1-13, 2-1, 2-6

- As writable data segment, 2-7

- Automatic creation, 3-47

- Default organization, 3-48

- Descriptor for, 2-6

- Overriding, 2-9

- Entry-list, 3-50

- Null (first) descriptor, 3-48

- Overwriting, 2-7

- Table-alias descriptor, 3-48

LOCATION syntax, 3-27, 3-43

Logical address, 1-8, 2-3

M

[NO]MAP control, 2-1, 4-32, 5-4, 5-8

- Example, 4-32

Map file, 4-6

MAP386, 1-1, 2-12, 4-21

Memory, 1-14

- Allocation, 1-14, 2-2, 3-23, 4-27, 4-30

- Sequence, 3-23

- Range, 1-14, 2-2

- Read/write, 1-14

- Reserved, 2-2

MEMORY definition, 2-2, 2-5,
3-21 3-26
And NOBOOTLOAD control, 3-21,
3-24
Example, 3-25 3-26
Ordering, 3-24
Memory management unit, 1-8
Memory mapping, 3-30
Memory model, 1-12
Flat (see also flat model), 1-12
Hybrid, 1-12
Paged, 1-12
Segmented, 1-12
MOD376 control, 2-2, 4-33
And BOOTSTRAP control, 4-33
And PAGETABLES control, 4-34
And PAGING definition, 4-34
And USE attributes, 4-34
Example, 4-34
MOD386 control, 4-33, 5-12
Example, 4-34

N

Near jump instruction, 4-15

O

[NO]OBJECT control, 4-35
Example, 4-36
OBJECT syntax, 3-52
Offset, 1-5, 1-8, 1-9
Output, 1-2, 2-13, 4-6, 4-35, 5-1, 5-2

P

P syntax, 3-27
Padding, 3-30, 3-37
Page directory, 1-14, 2-2
Creation, 2-10
Entry
Defaults, 3-30
Format, 3-29
Locked, 3-31
PAGE syntax, 3-33
Page tables, 1-14, 2-2, 2-10

Page tables (cont.)

Creation, 2-10
Default memory included, 3-29, 3-30
Entry
Defaults, 3-30
Format, 3-29
Locked, 3-31
PAGED syntax, 3-21
PAGETABLES control, 2-2, 4-37, 5-12
And BOOTLOAD control, 4-37
And MEMORY definition, 4-37
And MOD376 control, 4-37
And PAGING definition, 4-37
Example, 4-37
PAGETABLES syntax, 3-27
PAGING definition, 2-2, 2-10,
3-27 3-32
And MOD376 control, 3-27, 3-31
And PAGETABLES control, 3-27,
3-31
Examples, 3-31 3-32
Paging, 1-8
PARAGRAPH syntax, 3-33
_phantom_code, 6-2
_phantom_data_, 4-27, 4-28, 4-29, 6-2
Physical address, 1-8, 2-1, 2-2, 4-27
[NOT] PRESENT syntax, 3-15, 3-34,
3-44, 3-53
[NO]PRINT control, 2-1, 4-38
And [NO]OBJECT control, 4-38
And [NO]WARNINGS control, 4-39
And ERRORPRINT control, 4-39
Example, 4-39
Print file, 2-13, 4-6, 4-31, 4-32, 4-38,
5-1 5-13
Build program listing, 5-4
Gate table, 5-7
Header, 5-3
Page tables, 5-12
Segment map, 5-4
Task table, 5-8
Privilege level, 1-12
Privilege rules, 2-4

- Privileged instructions, 3-58
- Program development, 1-1
- Protection, 1-10, 1-13, 4-27, 4-28, 4-33
 - Privilege checking, 2-4
 - Subsystem, 1-13
 - Visibility, 1-12, 2-11
- Public symbol, 1-13, 2-11, 4-4, 4-15, 4-21, 4-43

Q

- QWORD syntax, 3-33

R

- RANGE syntax, 3-21
- [NOT] READABLE syntax, 3-34
- Reference resolution, 2-1, 2-11, 4-2, 4-3, 4-4
- Registers
 - 32-bit, 1-5
 - EFLAGS, 3-57, 3-58
 - I/O privilege level bit, 3-57, 3-58
 - Interrupt enable bit, 3-58
 - Virtual mode bit, 3-57, 3-58
 - General, 1-6, 1-9
 - Hidden cache, 1-8
 - Instruction, 1-6
 - Segment, 1-6, 1-7, 1-9
 - Loading, 1-7
 - Status, 1-6
 - System address, 1-6, 1-8, 1-9, 1-11
 - GDTR, 1-6
 - IDTR, 1-6
 - LDTR, 1-6
 - TR, 1-6, 1-11
- RELDESC control, 2-2, 2-12, 4-40
 - And NOBOOTLOAD control, 4-40
 - Example, 4-40
- Relocation information, 4-40
- Requested privilege level, 1-7
- RESERVE syntax, 3-21, 3-43
- Reset, 6-3
- RESET syntax, 3-27
- Reset vector, 4-33

- RO (see segment, read-only)
- ROM, 2-6
- RW (see segment, read/write)
- RW syntax, 3-27

S

- Segment
 - Access rights, 1-7, 2-1
 - Descriptor privilege level, 1-7
 - Present, 1-7
 - Type, 1-7
 - Attributes, 1-5, 1-13, 3-8
 - Combining, 1-13, 2-3, 3-34, 3-35, 4-28, 4-30
 - Conforming, 1-12, 2-4, 2-11, 3-35
 - Exporting, 2-2
 - Data/stack combined, 2-3, 2-8, 3-10, 3-35, 3-36, 3-37, 3-56, 4-29
 - Organization, 3-38, 3-57
- Defaults
 - Overriding, 3-41
- Execute-only, 3-23
- Execute/read, 3-23, 3-37
- Expanddown, 3-35, 3-36, 3-56, 4-28, 4-29
- Limit, 2-1
 - Granularity, 3-36
- Non-executable, 2-8, 3-10, 3-28
- Non-expanddown, 3-28, 3-56, 4-28
- Read-only, 3-22, 3-23, 3-37
- Read/write, 2-6, 3-7, 3-8, 3-22, 3-23, 3-28, 3-37, 3-43
- Stack, 2-3, 2-8, 3-10, 3-56, 4-28, 4-29
- Uninitialized areas of, 2-2, 4-25

- SEGMENT definition, 2-1, 2-5, 2-8, 3-33 3-42
- Examples, 3-42
- Segment descriptor, 1-6, 1-8, 1-9, 1-10, 2-6, 2-8, 3-33
- Accessed bit, 3-35
- Base, 3-35
- B/D-bit, 3-35, 3-36, 3-41
- USE16, 3-41

- Segment descriptor
 - B/D-bit (cont.)
 - USE32, 3-41
 - Default, 2-8
 - Overriding, 2-8
 - E/C-bit, 3-35
 - Format, 1-7, 3-35
 - G-bit, 3-35, 3-36
 - Granularity bit, 1-7
 - Limit, 3-35
 - Present bit, 3-35
 - Syntax summary, 3-40
 - Uninstalled, 2-7, 3-47
 - W/R-bit, 3-35
- Segmentation
 - Overriding, 2-1
- *SEGMENTS syntax, 3-21, 3-33
- Selector, 1-7, 1-8, 1-9, 1-10, 1-11
 - Format, 1-7
 - Requested privilege level, 1-7
 - Table indicator, 1-7
- SET syntax, 3-27
- Small model, 3-35
- SPECLN, 3-31, 3-38, 3-39
- SPECLN syntax, 3-33
- STACKS syntax, 3-52
- Subsystem, 1-12
- SYMBOL syntax, 3-10
- SYS\$OUTPUT, 4-4, 4-24
- System
 - Cold-start, 1-2
 - Configuration, 1-1
 - Initial state, 2-1
 - Static, 1-1
- System descriptor, 2-8
 - Data segment, 2-9
 - LDT descriptor, 2-8
 - Task descriptor, 2-8
- System development, 1-12
 - BLD386, 1-12
 - BND386, 1-12
- System interface, 1-13
- System software, 1-13

T

- TABLE definition, 2-1, 2-5, 2-7, 2-9, 3-43 3-51
 - Examples, 3-50 3-51
- *TABLES syntax, 3-21
- Task, 1-10, 1-11, 2-2
 - Definition, 2-11
 - Initial, 3-56
 - Initial registers, 3-53, 3-54
 - Protection, 2-11
- TASK definition, 2-1, 2-2, 2-5, 2-9, 2-10, 3-52 3-61
 - Examples, 3-60 3-61
- Task descriptor, 1-11, 2-6
 - Creation, 3-53
 - Defaults
 - Overriding, 2-9, 3-52
 - Installing, 3-53
 - Syntax summary, 3-60
- Task state segment, 1-10, 1-11, 1-14, 2-2, 2-10
 - 80286, 3-52
 - 80386, 3-52, 3-57
 - Automatic creation, 2-10
 - Back link, 1-11, 3-55
 - Base address
 - And BOOTLOAD control, 3-56
 - Overriding default, 3-56
- EFLAGS
 - IF, 3-58
 - IOPL, 3-58
 - VM, 3-58
- Format, 1-11, 3-55
- Initial values, 2-10
 - Code, data, and stack segments, 1-14
 - Status of registers and flags, 1-14
- I/O permission base, 1-11, 3-55
- Limit
 - Default, 3-57
 - Maximum, 3-57
- Syntax summary, 3-59
- T-bit, 1-11, 3-55, 3-56

- Task state segment
 - T-bit (cont.)
 - Default, 3-56
- Task switch, 1-7, 1-10, 3-56
- TASK syntax, 3-15
- *TASKS syntax, 3-21
- TITLE control, 4-41
 - DOS delimiters, 4-41
 - Example, 4-42
 - VMS delimiters, 4-41
- Translators, 1-1, 2-13, 3-41, 4-21, 5-1
- TRAP syntax, 3-15
- TSS (see task state segment)
- TSS descriptor (see task descriptor)
- Type checking, 2-2, 2-11, 4-43
- [NO]TYPE control, 2-2, 4-43
 - Example, 4-43

U

- UD1 syntax, 3-27
- UD2 syntax, 3-27
- UD3 syntax, 3-27
- US syntax, 3-27
- USE16 attribute, 2-1, 3-36, 3-37
 - And MOD376 control, 3-41
- USE16 syntax, 3-15, 3-34
- USE32 attribute, 2-1, 2-3, 3-10, 3-36, 3-37
 - And MOD376 control, 3-36
- USE32 syntax, 3-15, 3-34
- USREAL attribute, 2-1, 2-3, 2-5, 3-36, 3-48
 - And BOOTLOAD control, 3-36
 - And MOD376 control, 3-41
- USREAL syntax, 3-34

V

- Virtual mode
 - And BOOTLOAD control, 3-58
- [NOT] VIRTUALMODE syntax, 3-53

W

- Warning message, 4-23, 4-24, 4-44, 5-13
- [NO]WARNINGS control, 2-1, 4-4, 4-44, 5-13
 - And ERRORPRINT control, 4-44
 - Example, 4-45
- WC syntax, 3-15
- WORD syntax, 3-33
- Wrap, 2-2, 2-5, 3-23, 4-27, 4-33
- [NOT] WRITABLE syntax, 3-34



WE'D LIKE YOUR OPINION

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future versions.

To order publications, contact the Intel Literature Department (see page ii of this manual).

Fill in the squares below with a rating of 1 through 10:

POOR

AVERAGE

EXCELLENT

1

2

3

4

5

6

7

8

9

10

☐

Readability

☐

Technical depth

☐

Technical accuracy

☐

Usefulness of material for your needs

☐

Comprehensibility of material

☐

OVERALL QUALITY OF THIS MANUAL

If you gave a 4 or less (in any category), please explain here:

What suggestions would you have for improving this manual:

★ ★ ★ ATTENTION ★ ★ ★

Receive 50% off on the next Intel publication you buy. Send us your comments, and we'll send you a 50%-off certificate.

If you would like us to call you for more specific suggestions about this book, please additionally fill in your phone number below.

Name

Phone Number (

)

Address

Thanks for taking the time to fill out this form.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

**INTEL CORPORATION
DTO TECHNICAL PUBLICATIONS HF2-38
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978**



WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States, use the preprinted address provided on this form to return your comments. No postage is required. If you are not in the United States, return your comments to the Intel sales office in your country. For your convenience, a list of international sales offices is provided on the back cover of this document.

When finished, fold the form in half, tape it together, and mail it.

Please do not use staples.

Thanks for your comments.



International Sales Offices

BELGIUM

Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK

Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND

Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND

Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE

Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL

Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY

Intel Corporation S.P.A.
Milanfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

NETHERLANDS

Intel Semiconductor (Nederland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY

Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN

Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN

Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND

Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY

Intel Semiconductor GmbH
Seidlestrasse 27
D-8000 Muenchen 2



SERVICE INFORMATION



For service or assistance with Intel products, call:

- **1-800-INTEL-4-U** (1-800-468-3548) — in the United States and Canada
- Your local Intel sales office — in Europe or Japan
- Your Intel distributor — in any other area

Intel stands behind its products with a world-wide service and support organization. If you have problems, need assistance, or have a question, Intel can provide:

- On-site or carry-in service for hardware products
- Phone support for all Intel products
- On-site consulting for designing with Intel products or using Intel products in your designs
- Customer training workshops
- Updates to software products

To get more information on these services or to take advantage of them, call the INTEL-4-U number above.

All Intel products have a minimum warranty of 90 days, and all warranties include one or more of the services listed above. Talk with your Intel salesperson or call the INTEL-4-U number to determine the warranty services available for this product and how to register for them.

Intel is committed to continuing service for all its products.



INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95052-8130 (408) 987-8080

Printed in U.S.A.

DEVELOPMENT TOOLS AND SOFTWARE